BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

XP-002155044 SmartUpdat Dev lop r's Guid

This document describes the SmartUpdate feature of Netscape Communicator, as well as the AutoUpdate feature of Netscape Mission Control Desktop. SmartUpdate technology provides a way to automatically and securely install software on a user's machine. Although any software can be installed using SmartUpdate, SmartUpdate technology is ideally suited for installing Navigator plug-ins and Java classes.

Contents

Index

Table of Contents | Previous | Next | Index

Last Updated: 03/11/99 11:37:13

Chapter 1
Ov rview

In this chapter:

- Why Use SmartUpdate Technology?
- SmartUpdate Technology
- Where to Go from Here

The SmartUpdate technology described in this guide enables the SmartUpdate feature of Netscape Communicator, as well as the AutoUpdate feature of Netscape Mission Control Desktop. SmartUpdate technology provides a way to automatically and securely install software on a user's machine. Although any software can be installed using SmartUpdate, SmartUpdate technology is ideally suited for installing Navigator plug-ins and Java classes.

SmartUpdate technology uses Netscape's security framework to validate the source of the software. Users can decline the updating software on their machines, but intranet system administrators can configure Netscape Communicator so that users are not presented with the option of updating.

Developers can publish their software packages using SmartUpdate technology. Content creators can then modify their pages to initiate an installation through SmartUpdate. Users can easily and securely download and install r update software on their machines. In addition, system administrators can use AutoUpate to automatically upgrade software on their users' machines.

The primary audience for this manual is software developers who want their software to be installable through SmartUpdate. The manual describes how SmartUpdate technology works at runtime and what you do to mak that happen.

> **NOTE:** From the perspective of this underlying technology, AutoUpdate is a variant of SmartUpdate that serves a particular need of system administrators. This manual describes SmartUpdate, but everything that is relevant to SmartUpdate is also relevant to AutoUpdate.

Why Use SmartUpdate Technology?

Th re are many mechanisms for distributing and installing software over the Internet and over intranets. What is the advantage of using SmartUpdate technology?

The answer depends on who you are and your goals. For example:

- If you're an IS administrator, you probably need a way to easily and consistently distribute software to the desktops of all of your users. You might choose to package software for delivery with SmartUpdate and set up distribution with AutoUpdate or send an email message to all affected users, and have them click a link to install the software.
- If you're a content developer who needs a plug-in for your page to work properly, you've probably had to d al with users who don't have the plug-in or who have the wrong version. These users typically lose interest in your site rather than locating and downloading the missing plug-in. SmartUpdate technology gives you control ov r the plug-in installation process by using a script that triggers a SmartUpdate installation.
- If you're a software developer, you will appreciate how SmartUpdate technology eases the task of writing installation scripts. In addition, SmartUpdate installation scripts are reusable, so your first installation script can serve as a model for your next installation script. Easy-to-use installation scripts ease the workload of cont nt developers, thereby enticing more of content developers to use your software.

SmartUpdate Technology

SmartUpdate t chnology lev rag s xisting technologies as much as possible. The main techn logies you should b familiar with b for using SmartUpdate t chnology are:

- **JAR files:** JAR (Java archive) is an internet standard for cr ating fil archives. A SmartUpdate JAR fil is a c mpressed archive, containing files, security information about the files, and other "metainformation" about the

files. Ultimately, a SmartUpdate occurs when an installable JAR file is downloaded to the user's machine. (Note that a JAR file does not have to contain Java classes. This format can be used to package any files you want.) See "Creating and Signing a SmartUpdate JAR File" on page 41 for information on JAR files.

Object Signing: Object Signing is Netscape's security framework. It allows users to get reliable information about code they download in much the same way they can get reliable information about shrink-wrapped software. You don't need to understand the details, but it helps to have an overview. *Netscape Object Signing* provides such an overview; *Overview of Object-Signing Resources* provides information on related documentation.

- **JavaScript**: JavaScript is Netscape's cross-platform, object-based scripting language. To create an installable JAR file for SmartUpdate, you (the software developer) must write an installation script in JavaScript. JavaScript is also the language that a content developer would use to trigger a SmartUpdate installation from a web page. For information on using JavaScript, see the *JavaScript Guide*.

- **LiveConnect**: SmartUpdate technology extends JavaScript with a set of Java classes that provide the special capabilities (such as placing files on the user's disk) needed by trigger scripts and installation scripts. The Java classes are available to SmartUpdate scripts through JavaScript's LiveConnect functionality. Using LiveConnect is very simple, and is described in the *JavaScript Guide*.

- **Client Version Registry**: Netscape has created a cross-platform registry that records all the software installed through SmartUpdate. The Java classes provided by SmartUpdate technology take advantage of this registry. What you need to know about the registry is covered in this manual.

In addition to these technologies, tools are available to make it easy to use SmartUpdate:

- **SmartUpdate Builder**: SmartUpdate Builder is the Mission Control Desktop tool that makes it easy to create JAR files.

- **Communicator Upgrade Kit**: This kit, which comes with Mission Control Desktop, includes JAR files for using SmartUpdate technology to upgrade Communicator 4.0 to Communicator 4.5.

- **Netscape Signing Tool**: The Netscape Signing Tool is used to sign JAR files with the digital ID of the generator of the JAR file.

What's New in SmartUpdate Technology?

Communicator 4.5 supports several new methods for the SoftwareUpdate and Trigger objects that make it easier to write installation scripts. For a complete list of the changes, see Appendix C, "Release Notes." For detailed information, see Chapter 7, "Reference."

Where to Go from Here

This manual contains the following chapters and appendixes:

- Chapter 2, "A Quick Look"

  Provides a simple example from start to finish, showing all the steps you need to take to prepare your software package for SmartUpdate.

- Chapter 3, "SmartUpdate at Runtime"

  Describes the typical user experience and discusses briefly how you can change this.

- Chapter 4, "Writing an Installation Script"

  Provides details on how to write an installation script for a JAR file. (You only need to read this chapter if you're creating the JAR file. You don't need to if you're writing the trigger page that downloads the archive.)

- Chapter 5, "Packaging Software"

  Describes the process of putting the software and installation script together into an installable JAR file. (You only need to read this chapter if you're creating the JAR file. You don't need to if you're writing the trigger page that downloads the archive.)

- Chapter 6, "Initiating a SmartUpdate Installation"

Provides details on how to initiate a software download and installation through SmartUpdate. (You need to read this chapter if you're writing the trigger page that downloads a JAR file. If you're creating the JAR file, you can skim this material.)

- Chapter 7, "Reference"

  Contains reference information on the special Java classes you use for writing trigger and installation scripts.

- Appendix A, "Sample Installation Script"

  Contains an example of a complex installation script.

- Appendix B, "End User Problems"

  Describes typical problems end users might encounter using SmartUpdate and what should be done about them.

- Appendix C, "Release Notes"

  Contains information on known problems with SmartUpdate technology. Before you start working with SmartUpdate technology, be sure to read the release notes.

- Appendix D, "The NSDiff Utility"

  Contains information on using the NSDiff utility, which is used in conjunction with the `Patch` method.

Chapter 2
A Quick Look

In this chapter:

- Write Your Software
- Write an Installation Script
- Create a JAR File
- Write the Trigger Script
- Publish the Trigger Script

This chapter provides a simple example from start to finish. It lists but does not explain all the steps that need to be performed to prepare and deliver software using SmartUpdate technology. For an explanation of each step, you'll need to read the other chapters of this manual.

This example prepares a plug-in for SmartUpdate:

- The name seen by the user for the plug-in is "Royal Airways Plug-in"
- The version is 3.2.1.0.
- The Client Version Registry name for the package is `plugins/royalairways/RoyalPI/`.
- The installation places the files `rplugin.exe`, `NPRPI.DLL`, and `help.htm` into the `Royalairways` subdirectory under the `Plugins` folder.

Writ Your Software

With the exception of Mac OS platforms, there are no special requirements for writing SmartUpdate installable software. The only special requirement for software that is to be installed on Mac OS computers is that the software files must be converted to AppleSingle format before they are placed in the JAR file. You can use a program such as F tch or Stuffit Deluxe to convert your files to this format.

Write an Installation Script

You can write your entire installation process in the JavaScript installation script or you can write a minimal JavaScript installation script that launches an external installer. Chapter 4, "Writing an Installation Script," discusses these choices.

Here's a sample installation script for the Royal Airways plug-in. The functions used in this script are documented in Chapter 7, "Reference."

```
// Conditional alert.
function cAlert (message) {
   if (!this.silent)
      alert(message);
}

// Conditional confirm.
function cConfirm (message) {
   if (this.silent)
      return true;
   else
      return confirm(message);
}

// Variable indicating whether or not installation should proceed.
bInstall = true;

// Make sure Java is enabled before doing anything else.
```

```
if ( !navigator.javaEnabled() ) {
   bInstall = false;
   cAlert ("Java must be enabled to install.");
}

// Make sure installation is attempted on correct machine architecture.
else if ( navigator.platform.compareTo("Win32") != 0 ) {
   bInstall = false;
   cAlert ("This plug-in only runs on Win32 platforms.");
}

// Check user-interface language, if appropriate.
else if ( navigator.languge.compareTo("en") != 0 ) {
   bInstall = cConfirm("This plug-in uses the English language.
      You do not appear to be using English on this machine.
      Install anyway?");
}

// If all conditions look good, proceed with the installation.
if (bInstall) {
   // Create a version object and a software update object
   vi = new netscape.softupdate.VersionInfo(3, 2, 1, 0);
   su = new netscape.softupdate.SoftwareUpdate(this,
      "Royal Airways Plug-in");

   // Start the install process
   err = su.StartInstall("plugins/royalairways/RoyalPI/", vi,
      netscape.softupdate.SoftwareUpdate.FULL_INSTALL);

   if (err != 0)
      cAlert ("Installation error. Aborting.");

   else {
      bAbort = false;

      // Find the plug-ins directory on the user's machine
      PIFolder = su.GetFolder("Plugins");

      // Install the files. Unpack them and list where they go
      err = su.AddSubcomponent("RoyalPI Executable", vi, "rplugin.exe",
         PIFolder, "RoyalPI/rplugin.exe", this.force);
      bAbort = bAbort || (err !=0);

      if (!bAbort) {
         err = su.AddSubcomponent("RoyalPI DLL", vi, "NPRPI.DLL",
            PIFolder, "", this.force);
         bAbort = bAbort || (err !=0);
      }

      if (!bAbort) {
         err = su.AddSubcomponent("RoyalPI Help", vi, "help.htm",
            PIFolder, "RoyalPI/help.htm", this.force);
         bAbort = bAbort || (err !=0);
      }

      if (bAbort)
         cAlert ("Installation error. Aborting.");
}

// Unless there was a problem, move files to final location
// and update the Client Version Registry
if (bAbort) {
   cAlert ("Installation error. Aborting.");
   su.AbortInstall();
}
else {
   err = su.FinalizeInstall();
```

```
        // Refresh list of available plug-ins
        if (err == 0)
           navigator.plugins.refresh(true);
        else if (err == 999) {
           cAlert("You must reboot to finish this installation.");
           err = 0;
        }
     }

     if ( bAbort || err != 0 )
        cAlert("Install encountered errors.");
}
```

Cr ate a JAR File

After you have written your software and an installation script, package your software, its installation script, and any auxiliary files your software needs into a JAR file.

Before you can create your JAR file, you must have a code-signing certificate. You use this certificate to sign your code and to identify yourself to the users of your installation. Once you have your code-signing certificate, you use it to package your software and the installation script into an archive. You use the Netscape Signing Tool to create a signed and installable JAR file. If you have Mission Control Desktop, you can use SmartUpdate Builder to create JAR files.

Write the Trigger Script

Once an installable JAR file has been published on the Internet or on your company's intranet or extranet, end us rs and content developers can access it either by clicking a link to the SmartUpdate JAR or through a trigger script. If access is through a trigger script, the script can perform various checks to ensure that the correct JAR file is downloaded and installed on the user's machine.

The following HTML page lets a user click a button and run a trigger script that initiates SmartUpdate for the JAR file named royalpkg.jar.

This trigger script performs checks that duplicate checks performed in the installation script. Putting the checks in the trigger script prevents users from wasting time downloading archives they don't need. The installation script double checks to prevent installing useless or even harmful files in case the user bypassed the trigger script when he or she downloaded the JAR file.

```
<HTML>
<HEAD>
<SCRIPT>

function downloadNow () {

if ( navigator.javaEnabled() ) {

   trigger = netscape.softupdate.Trigger;
   if ( trigger.UpdateEnabled() ) {

      if (navigator.platform == "Win32") {
         vi = new netscape.softupdate.VersionInfo(3, 2, 1, 0);
         trigger.ConditionalSoftwareUpdate(
            "http://royalairways/royalpkg.jar",
             "plugins/royalairways/RoyalSW",
             vi, trigger.DEFAULT_MODE);
      }

      else alert("This plug-in only runs on Windows NT/95.")
   }

   else
      alert("Enable SmartUpdate before running this script.");
   }

else
```

```
      alert("Enable Java before running this script.");

</SCRIPT>
</HEAD>

<BODY>

<H1>Royal Airways Download Page</H1>
<P>Welcome! From this page you can download the Windows 95/NT version of
our famous plug-in. To do so, simply click the button.</P>

<CENTER><FORM METHOD="Post">
<INPUT TYPE="button" VALUE="Download Plug-in Now!"
   onClick="downloadNow();">
</FORM></CENTER>

</BODY>
</HTML>
```

Publish the Trigger Script

Publish the trigger script written in the last section on your web site.

Make sure that your web server is configured to serve JAR files with the MIME type `application/java-archive`.

Chapter 3
SmartUpdate at Runtime

In this chapter:

Enabling SmartUpdate

Initiating SmartUpdate

Installing the Software

A SmartUpdate consists of two phases:

- Initiating the process with a trigger and locating the archive
- Downloading the JAR file, getting permission to do the installation, and installing the software

This chapter gives an overview of what happens at runtime for both phases. Subsequent chapters supply the details on the work that must be done before a SmartUpdate process can be initiated.

Enabling SmartUpdate

By default, SmartUpdate is enabled for Communicator. However, whether SmartUpdate is enabled is controlled by a user preference on the Advanced panel of the Preferences dialog. If Enable SmartUpdate is checked, SmartUpdate can download and install software on that computer, subject to the usual security checks. If it is not checked, SmartUpdate cannot be used on that computer.

Having Enable SmartUpdate checked corresponds to the following line in the user preferences file:

```
user_pref("autoupdate.enabled",true);
```

Initiating SmartUpdate

Typically, a user initiates a SmartUpdate process by visiting a page and clicking a link that triggers an installation. The intrusiveness of the SmartUpdate process depends on the computer's security settings and details of the trigg r script.

> **NOTE:** Although content providers must provide the trigger that downloads software, you (the softwar provider) should supply content providers with a sample script and give them the information they need to modify the script, if appropriate. In addition, you may want to have a download page of your own that initiates SmartUpdate for your software. To get an idea of the information you should supply to content developers, see *SmartUpdate for Content Developers*.

The SmartUpdate technology does not provide a user interface for initiating SmartUpdate. It is up to the content provider to provide this interface. A SmartUpdate may require no user interface at all, in which case the downl ad and installation occurs without user intervention. A minimal user interface may be to "Click here to download th latest KillerApp."

SmartUpdate technology provides tools to create "smarter" download pages to initiate a SmartUpdate process. For example, the page could detect the software the user has already installed and only perform an upgrade if an upgrade is appropriate.

Ultimately, an HTML page triggers a SmartUpdate by containing a Java or JavaScript statement that explicitly requests the s ftware or by containing an explicit link to a JAR file. See Chapter 6, "Initiating a SmartUpdate Installation," for details about SmartUpdat techn logy triggers.

Whatev r the trigger, Communicator downloads the specified Jar file into a t mporary download ar a. Th downl ad d file must b in JAR format and must have a JavaScript installation script. JAR format allows you to bundl files with an installati n script and security informati n.

Installing the Software

Once the JAR file is in the download area, the installation script starts to run. It runs in three phases:

- Setup
- Getting permission to install
- Installing files and cleaning up the temporary file location

The following sections describe the three phases.

Installation Setup

Your installation script may do some initial checking before it creates a `SoftwareUpdate` object. These checks ensure that the environment is appropriate for installing the JAR file.

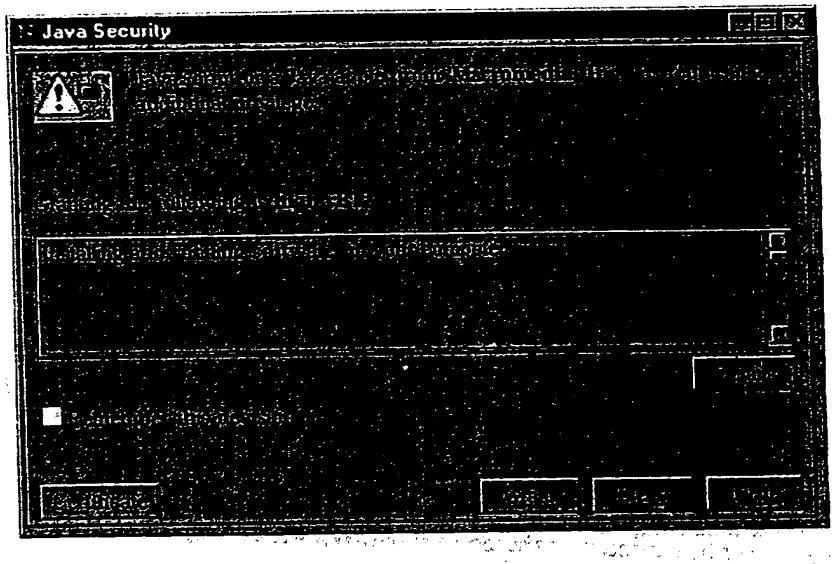
Getting Permission

Communicator must acquire permission to install software for the first time or to update existing software. This permission is given on the basis of a Certificate. If SmartUpdate has previously been used to install software on the user's computer that you developed and the user checked the "Remember this decision" box during the installation, the security dialog box will not appear when the user installs software signed with the same Certificate.

Installing and updating software require the ability to write to the user's hard disk, possibly overwriting existing files and executing an arbitrary executable. A malicious individual with permission to write to the user's hard drive could cause extreme damage. For that reason, the installation script must be signed so the user can verify the source of the new software.

When SmartUpdate is given a JAR file, it extracts the installation script from the JAR file and runs it. By default, the first thing the user sees is the dialog box shown in Figure 3.1. This dialog box appears when the installation script calls the `StartInstall` method. You can post your own dialog box before this dialog box, but you cannot prevent this dialog box from appearing.

Figure 3.1 Security dialog box.



NOTE: The Security dialog box does not appear if the user previously checked the "Remember this decision" box so that installation permission is permanently granted.

The dialog is part of the standard security interface. It asks the user for permission to run the installation script. If the user grants the installation script the necessary privileges, the rest of the installation script runs. If the user denies permission, the `StartInstall` method returns an error code and the installation script continues to run, but it cannot write to the user's hard disk.

As the software developer, it is your responsibility to provide the installation script. If appropriate, you can use custom HTML and JavaScript windows to add additional user interface. SmartUpdate technology provides new Java classes you can use in a JavaScript installation script to perform your installation. If

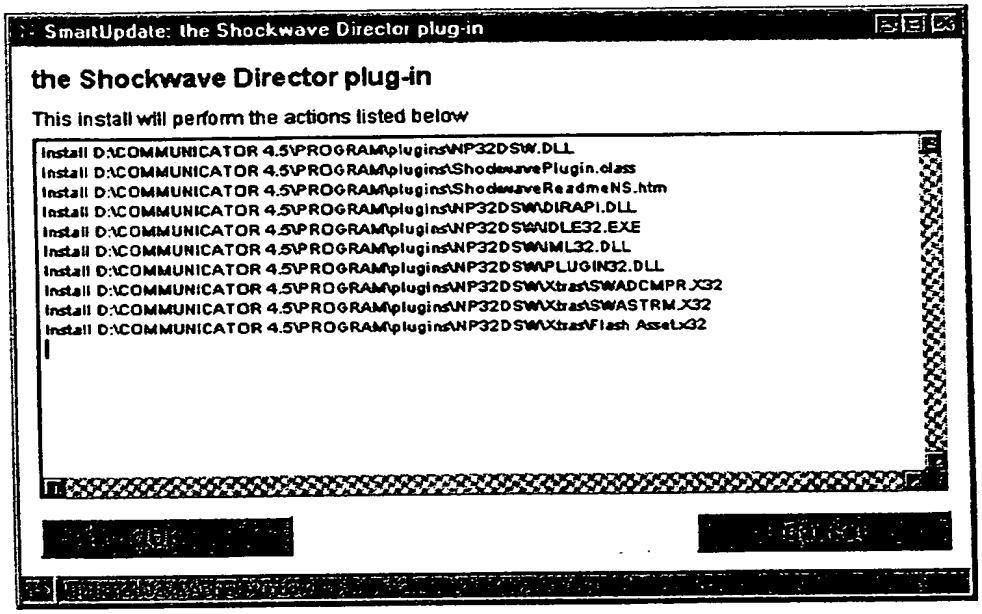
you use these classes, Communicator stores a record of the software in a cross-platform registry, called the Client Version Registry. For detailed information on what you can put in an installation script, see Chapter 5, "Packaging Software."

Installing the Files

If the user has the "Confirm" SmartUpdate preference set, once permission to install or update software has been granted, SmartUpdate immediately replaces the security dialog with the installation dialog box shown in Figure 3.2. The installation dialog box lists the names of the files that are to be installed.

When this dialog box first appears, the OK button is dimmed. When the installation script reaches its call to `FinalizeInstall`, the OK button becomes active. At this point, the user must click OK to proceed.

Figure 3.2 SmartUpdate information dialog box.



Once the user grants final permission, SmartUpdate completes the installation by moving the files from the temporary area to their final destinations.

If the user does not have the "Require manual confirmation of each install" SmartUpdate preference set, the dialog box shown in Figure 3.2 does not appear. Instead, the installation script runs to completion without requesting confirmation from the user.

More About Security

SmartUpdate technology uses the Netscape security framework embodied in object signing to provide security for SmartUpdate installations. Because of the high risk associated with writing to an user's hard disk, SmartUpdate technology requires you to sign your installable JAR archives and may post the appropriate security dialog boxes before it writes files to their final destinations. For information on Netscape's object signing capabilities, see *Netscape Object Signing*. This section gives a brief overview of the framework as it applies to SmartUpdate technology.

The first step in getting permission to install files is to decode the JAR file enough to see by whom its contents have been signed. The sequence of events thereafter depends on whether the site administrator has placed special security restrictions on the user's computer.

- If no special security restrictions have been applied, Communicator displays the standard security dialog box including information about who signed the JAR installation script and asking the user for permission to install the software. If the user says to continue, Communicator starts the actual installation. If the user says to stop, the `StartInstall` method returns an error code to the installation script, which should execute any cleanup code that it requires and exit. Communicator then deletes the downloaded JAR file.

- The site administrator can use Netscape Mission Control Desktop to place restrictions on the software that can be installed by means of SmartUpdate on a computer. The following are the general choices the site administrator can make regarding software installation:

- Never install any software signed by this entity. Do not display the standard security dialog box.
- Always install any software signed by this entity and do not display the standard security dialog box. Start the installation immediately and do not ask the use to confirm the installation.
- Never install additional software on this computer. Do not display the standard security dialog box.
- Let the user decide what to do in every case. Display the standard security dialog box. Before actually installing the files, display a list of file names that are going to be installed and ask the user for confirmation.

For more information about Mission Control Desktop, see the *Mission Control Desktop Administrator's Guide*, available online at http://www.insight.netscape.com.

Chapter 4
Writing an Installation Script

In this chapter:

What's Been Added for Writing Installation Scripts

What Your Installation Script Should Do

Positioning Software in the Client Version Registry

Example Installation Scripts

You need to write a JavaScript installation script to install your software. When Communicator downloads the JAR file, it locates the JavaScript installation script and executes it to start the installation process. You can either choose to write your entire install process in the JavaScript installation script or you can write a minimal JavaScript installation script that launches a native executable installer packaged in the SmartUpdate JAR file.

Although using an existing installer allows you to leverage past work, you may choose to write a new JavaScript installation script using the Java classes described in this manual. Writing a JavaScript installation script provid s a single cross-platform script—you do not have to write a separate script for each platform you support. In additi n, th JavaScript installation script lets you use the Client Version Registry to keep track of your software, tracking v rsions and locations installed on the computer. Lastly, for Communicator 4.5 and later, a JavaScript installation script allows the user to uninstall components that were installed through SmartUpdate.

This chapter describes how to write an installation script for SmartUpdate, either completely using JavaScript or using a minimal JavaScript script and a native executable installer.

Special Note for Java Classes

If the software you're installing is signed Java classes, there's a small wrinkle. Typically, signed Java classes ar served as needed from a web server. If, however, you want users to be able to download the signed classes to their computers and run them locally, you package the JAR file containing the Java classes into another signed JAR fil , this one with an installation script.

This manual discusses the outer JAR file containing an installation script. You must create separately the first JAR file that contains the signed Java classes but does not contain an installation script.

You should know two special things about using SmartUpdate to install signed Java classes:

- Typically, you register signed Java classes in the Java Download area of the Client Version Registry and install them in the Java Download area of the Communicator installation.
- When users install new signed Java classes, they must restart Communicator for the new class to be available. At the end of your installation script, you should post a message letting users know about this.

What's Been Added for Writing Installation Scripts

To help you write an installation script, several new Java classes have been added. The primary new classes are `netscape.softupdate.VersionInfo` and `netscape.softupdate.SoftwareUpdate` and are described in Chapter 7, "Reference." Through JavaScript's LiveConnect feature, these classes can be used with JavaScript and let you:

- Specify a version for your software
- Specify dir ctori s on the l cal computer
- Extract files fr m a JAR file onto th local disk
- Update the Client Version Registry

You can, of course, also use the full range of JavaScript to implement as complex an installation process as you need.

In addition, if you are working on a PC platform, you may need to also manipulate `.ini` files or the Windows Registry. To do so, you can use the `netscape.softupdate.WinProfile` or `netscape.softupdate.WinReg` classes described in Chapter 7, "Reference."

If you are working on a Mac OS platform, you have full access to the G stalt selectors.

Finally, the JavaScript `navigator` object has several properties, such as `language`, `platform`, and `appVersion`, that are useful when writing an installation script. For information on the `navigator` object and its properties, see the *JavaScript Guide*. For information on new JavaScript properties, see *What's New in JavaScript 1.3*.

What Your Installation Script Should Do

In general, your installation script should perform the tasks discussed in the following sections. Not all installation scripts need to perform all of these tasks, nor do all of them have to be done in the order presented here.

Your installation script runs in a special context; the `SoftwareUpdate` class can be used only in this context. In this special context, `this` contains pointers to the JAR file being installed and to other information the `SoftwareUpdate` obj ct needs. Your script can always use `this` to refer to the JavaScript context.

B sil nt if asked to

Your installation may want to communicate with the user by posting dialog boxes or other windows. The trigger script written by the content provider can request that you not display any information to the user.

Th cont nt developer indicates this by passing the appropriate parameter to the `ConditionalSoftwareUpdate` or `StartSoftwareUpdate` method of the `Trigger` object, as described in " Installing Silently."

When you write your installation script, the value of `this.silent` indicates whether or not this request was made in the trigger script. Your installation script should obey this request. That is, if `this.silent` is true, your script should not post any dialog boxes or display any other windows.

> **NOTE:** A silent installation can occur only if the signer of the JAR file has the `SilentInstall` privilege. This privilege can be set only using Netscape Mission Control.

You can us functions such as those that follow to check the value of `this.silent` before posting a message or asking a question:

```
// Conditional alert.
function cAlert (message) {
   if (!this.silent)
      alert(message);


// Conditional confirm.
function cConfirm (message) {
   if (this.silent)
      return true;
   else
      return confirm(message);
```

ll example installation scripts in this manual use these functions.

h ck Error Return Codes

Many of th methods you use can return error cod s. Your script should always check for th se rrors and handle h m appr priat ly when it encounters one.

) cide Wh th r It's Possible to Install

h first thing your installation script should do is check that Java is enabled on th user's computer. If it is disabled, t the user know by posting an alert and then discontinue the installation. In addition, if the installation is specific to a articular computer architectur or browser version, check that as well.

For example, if this installation script is specific to Windows 95/NT, you can make these checks with code similar to this:

```
if ( navigator.javaEnabled() ) {
   if ( navigator.platform == "Win32") ) {
      // ... perform actual installation...
      }
   else
      cAlert("This installation only works on Windows 95/NT.");
else
   cAlert("Java is not enabled; enable it and try again.");
```

You must check that Java is enabled before you can do anything with the Java classes discussed in "What's Been Added for Writing Installation Scripts" on page 26. Your installation script does not need to check whether SmartUpdate has been enabled; the script cannot be run if it is disabled.

Note that a well-written trigger script may have already checked this information. Checking in the trigger script ensures that users don't needlessly download JAR files they can't use. However, users may bypass a trigger script and directly download a JAR file. In this situation, these checks in the installation script protect against attempting an inappropriate installation.

Cr ate a Software Update Object and a Version Object

The software update object is your main tool for installing your software. When you create it, the first parameter is always the keyword `this`, indicating the JavaScript context. The second is a name for your package that is used in messages to the user. For example:

```
su = new netscape.softupdate.SoftwareUpdate(this,
   "Royal Airways Plug-in");
```

The version object specifies the version of the software you're installing. For example:

```
vi = new netscape.softupdate.VersionInfo(3, 2, 1, 0);
```

You have to create these objects before you can go any further.

D t rmine a Location in the Client Version Registry

The next step registers your software in the Client Version Registry. You need to be careful when you pick your software's location in the registry. "Positioning Software in the Client Version Registry" on page 33 discusses the things you need to know to choose a good location.

Start the Installation

You must call the `StartInstall` method of your `SoftwareUpdate` object to start the installation process. This method displays the security dialog to the user, providing the option to cancel the installation. A typical call looks like:

```
err = su.StartInstall("plugins/royalairways/RoyalPI/", vi,
   netscape.softupdate.SoftwareUpdate.FULL_INSTALL);
```

None of the other `SoftwareUpdate` methods work if you don't start the installation or if the user denies permission for the installation.

This call to `StartInstall` specifies three things:

- The location in the Client Version Registry to put information about the software. For information on how to appropriately choose a location, "Positioning Software in the Client Version Registry" on page 33.
- Version information for the software package as a whole.
- Wheth r to display the progress dialog box during the installation.

Sp cify Wh re th Piec s Go

You may want to install a plug-in in Communicator's plug-ins directory. You can call the `SoftwareUpdate` object's `GetFolder` m thod to determine the names of some standard directories you might use. For example:

```
PIFolder = su.GetFolder("Plugins");
```

Typically, if you're installing signed Java classes, you install them relative to the Java Download directory. In this case, you would find that directory with this code:

```
JavaFolder = su.GetFolder("Java Download");
```

You use the `AddSubcomponent` method to extract individual files from the specified dir ctory in the archive and install them in a temporary location. You call this method once for each file you need to install. Typically, these calls do not install files if a more recent version is already present. You can override this behavior, if you need to.

Downgrade a Component If Requested

Usually, SmartUpdate updates a component only if it is newer than a previously installed version of that component. In some circumstances, you may want to downgrade a component by installing it even if a previously installed version has a higher version number. You indicate this in your trigger script by passing the appropriate parameter to the `ConditionalSoftwareUpdate` or `StartSoftwareUpdate` method of the `Trigger` object, as described in "Downgrading a Package" on page 53.

When you write your installation script, the value of `this.force` indicates whether this request was made in the trigger script. Usually, your installation script should obey this request. To do so, all calls to the `SoftwareUpdate` object's `AddSubcomponent` method pass `this.force` as their last argument.

However, it may be inadvisable to downgrade one of the components of a software package, even if the package as a whole is being downgraded. For example, on Windows platforms, it is usually inadvisable to downgrade a shared component (such as most DLLs in the `Windows` and `Windows/System` directories). When you use `AddSubcomponent` to install a shared component, you should use `false` as its last argument.

Execute a Native Executable Installer

Do this step only if you're using a native executable installer. Don't do this if you're writing the entire installation script in JavaScript.

The `SoftwareUpdate` object's `execute` method allows you to execute a binary file located in the JAR file. The binary file can be an installer or any other application. The actual execution happens when you call the `FinalizeInstall` method. Calling `execute` puts the file in a temporary location and queues it to be executed. For example, the binary might be the self-extracting installer for a plug-in. Communicator will delete the binary later.

T ll th User If Rebooting is Necessary

This is an unusual situation, but it's very important to remember! Some installations replace a file currently in use or require a system reboot for some other reason. If this is true for your installation, tell the user.

Finaliz or Abort the Installation

Calling the `FinalizeInstall` method should always be the last call in the installation. After your script calls `FinalizeInstall`, it can perform other actions, such as displaying a README file.

In the following installation fragment, the `bAbort` variable indicates whether an error has been encountered up to this point.

```
if (bAbort) {
   cAlert("Installation aborted.");
   su.AbortInstall();
}
else {
   err = su.FinalizeInstall();
   if (err == 0)
      navigator.plugins.refresh(true);
   else if (err == 999)
      cAlert("You must reboot to finish this installation.");
   else
      cAlert("Error in installation. Aborted.");
}
```

If an error occurs, call `AbortInstall` to clean up the disk and remove the progress box.

If all went well, call `FinalizeInstall` at the end of the installation. This enables the Install button in the install progress dialog and gives the user a final chance to abort the installation. When the user clicks Install, Communicator moves the files to their permanent location and updates the Client Version Registry to reflect the installation.

The call to `navigator.plugins.refresh` is used only when installing a plug-in; it refreshes Communicator's list of available plug-ins. This is not required, but it is strongly recommended if you are installing a plug-in. This call efficiently updates pages that use this plug-in. The `refresh` method finds all instances of the Default plug-in whose MIME types are now handled by your plug-in. It then deletes these old instances and creates new instances for the updated plug-in.

If the `FinalizeInstall` method returns 999, the installation changed a file that is currently in use by the operating system. For the installation to be completed, the computer must be rebooted. Your installation script might check for this return code and display a warning to the user.

Positioning Software in the Client Version Registry

The Client Version Registry is a registry provided by Netscape on all platforms, separate from the Windows registry. This registry is a hierarchical description of the software registered for use with Communicator. When writing an installation script for SmartUpdate, one of the things you must decide is where to place your software within the Client Version Registry.

There are two main considerations in picking a registry key (that is, positioning software in the registry):

- Pick a name that is unlikely to also be picked by someone across the world with a similar idea to yours. (This is referred to as the *namespace problem*.)
- Decide whether or not the package you're installing is dependent on a particular installation of Communicator.

Namespace Clashes

The namespace problem is a common one when installing software. If two different SmartUpdate packages use the same registry name, problems occur that are similar to the problems that occur when two applications use the same Windows registry key—the applications read data set by each other and misinterpret it as their own.

The usual solution to this problem is to include your company name as part of your key. For example, the Windows registry encourages this for installing software on a Windows computer and Sun encourages this for picking package names for Java classes. The same approach is suitable for positioning software in the Client Version Registry.

SmartUpdate uses the registry data to determine when an update is necessary. If another package using the same registry key was installed before your package, two serious problems may occur. To illustrate these problems, assume the other package has a higher version number than your package:

- Installation of your package might never occur. In the normal installation mode, the Client Version Registry would believe your package was already installed and not reinstall it.
- Alternatively, if your installation process uses `FORCE_MODE` to force installation because you expect the registry to contain a lower version number for your package, your package would be installed properly. However, the next time a request came to install the other package, it would be needlessly downloaded and reinstalled because the Client Version Registry would now believe an earlier version of that package was on the computer.

Communicator Dependency

The other major question you should ask yourself is "If the user had *two* versions of Communicator installed, should a single SmartUpdate of my package install the software for both copies, or only for the currently running one?"

The most common case is that your software should be installed separately for each version of Communicator on the computer, because it may rely on features specific to a particular version. In this case, you should use a *relative* pathname to mark the installation in the registry as relative to the currently running Communicator.

In particular, software installed in the `plugins` directory or in the `java/download` directory is treated by Communicator as relative to the Communicator version. Therefore, if you install software in these directories, you should also make its entry in the registry be relative.

If your software is usable by multiple versions of Communicator or is not relevant to Communicator at all, you can use an *absolute* pathname to specify the registry position.

An absolute registry name starts with a slash, as in:

```
/royalairways/RoyalStandAlone
```

A relative name doesn't start with a slash. In addition, you should make your entry relativ to the `plugins` area or the `java/download` area of the r gistry. Register a plug-in in th `Plugins` area, as in:

```
plugins/royalairways/RoyalPI
```

Register signed Java classes in the `java/download` area, as in:

```
java/download/royalairways/RoyalJava
```

Example Installation Scripts

This section provides three sample installation scripts. These samples illustrate installing a plug-in using a nativ executable installer and installing a plug-in or signed Java classes without a native executable installer.

Using a Native Executable Installer

This section describes a simple installation script that uses a native executable installer to do most of the installation. If you already have, for example, an InstallShield installer for your software, you could package that installer for use with SmartUpdate. For this example:

- The package name is "Royal Software."
- The version is 2.1.0.0.
- The Client Version Registry name for the package is `plugins/royalairways/RoyalSW/`. This is in the plug-ins area, relative to the Communicator installation.
- The installation calls `royalpkg.exe` to perform the actual installation.

Th complete script follows:

```
// Make sure Java is enabled before doing anything else.
if ( navigator.javaEnabled() ) {

    // Create a version object and a software update object
    vi = new netscape.softupdate.VersionInfo(2, 1, 0, 0);
    su = new netscape.softupdate.SoftwareUpdate(this, "Royal Software");

    // Start the install process
    su.StartInstall("plugins/royalairways/RoyalSW", vi,
        netscape.softupdate.SoftwareUpdate.FULL_INSTALL);

    // Unpack the native installer found in the JAR file to
    // a temporary location
    su.Execute("royalpkg.exe");

    // Perform requested execution and update the
    // Client Version Registry
    su.FinalizeInstall();
```

Installing a Plug-in from JavaScript

This section describes a simple JavaScript installation procedure for a plug-in. For this example:

- Th packag name is "Royal Airways Plug-in."
- The version is 3.2.1.0.
- The Client Version Registry name for the package is `plugins/royalairways/RoyalPI/`.

- The installation places the files `rplugin.exe`, `NPRPI.DLL`, and `help.htm` into the `RoyalAirways` subdirectory under th `Plugins` folder.

The complete script follows:

Here's a sample installation script for the Royal Airways plug-in:

```
// Conditional alert.
function cAlert (message) {
   if (!this.silent)
      alert(message);
}

// Conditional confirm.
function cConfirm (message) {
   if (this.silent)
      return true;
   else
      return confirm(message);
}

// Variable indicating whether or not installation should proceed.
bInstall = true;

// Make sure Java is enabled before doing anything else.
if ( !navigator.javaEnabled() ) {
   bInstall = false;
   cAlert ("Java must be enabled to install.");
}

// Make sure installation is attempted on correct machine architecture.
else if ( navigator.platform.compareTo("Win32") != 0 ) {
   bInstall = false;
   cAlert ("This plug-in only runs on Win32 platforms.");
}

// Check user-interface language, if appropriate.
else if ( navigator.languge.compareTo("en") != 0 ) {
   bInstall = cConfirm("This plug-in uses the English language.
      You do not appear to be using English on this machine.
      Install anyway?");
}

// If all conditions look good, proceed with the installation.
if (bInstall) {
   // Create a version object and a software update object
   vi = new netscape.softupdate.VersionInfo(3, 2, 1, 0);
   su = new netscape.softupdate.SoftwareUpdate(this,
      "Royal Airways Plug-in");

   // Start the install process
   err = su.StartInstall("plugins/royalairways/RoyalPI/", vi,
      netscape.softupdate.SoftwareUpdate.FULL_INSTALL);

   if (err != 0)
      cAlert ("Installation error. Aborting.");

   else {
      bAbort = false;

      // Find the plug-ins directory on the user's machine
      PIFolder = su.GetFolder("Plugins");

      // Install the files. Unpack them and list where they go
      err = su.AddSubcomponent("RoyalPI Executable", vi, "rplugin.exe",
         PIFolder, "RoyalPI/rplugin.exe", this.force);
      bAbort = bAbort || (err !=0);
```

```
    if (!bAbort) {
        err = su.AddSubcomponent("RoyalPI DLL", vi, "NPRPI.DLL",
            PIFolder, "", this.force);
        bAbort = bAbort || (err !=0);
    }

    if (!bAbort) {
        err = su.AddSubcomponent("RoyalPI Help", vi, "help.htm",
            PIFolder, "RoyalPI/help.htm", this.force);
        bAbort = bAbort || (err !=0);
    }

    if (bAbort)
        cAlert ("Installation error. Aborting.");
}

// Unless there was a problem, move files to final location
// and update the Client Version Registry
if (bAbort) {
    cAlert ("Installation error. Aborting.");
    su.AbortInstall();
}
else {
    err = su.FinalizeInstall();

    // Refresh list of available plug-ins
    if (err == 0)
        navigator.plugins.refresh(true);
    else if (err == 999) {
        cAlert("You must reboot to finish this installation.");
        err = 0;
    }
}

if ( bAbort || err != 0 )
    cAlert("Install encountered errors.");
}
```

Installing Signed Java Classes

This section describes a simple JavaScript installation procedure for installing a JAR file containing signed Java classes. For this example:

- The name for the set of Java classes is "Royal Java."
- The version is 1.0.
- The Client Version Registry name for the classes is `Java/Download/royalairways/RoyalJava/`. This is relative to the Java Download area of the Communicator installation.
- The installation places the JAR file containing the signed Java classes into the `Java/Download` folder.

The complete script follows:

```
    no cAlert();

// Make sure Java is enabled before doing anything else.
if ( navigator.javaEnabled() ) {

    // Create a version object and a software update object
    vi = new netscape.softupdate.VersionInfo(1, 0, 0, 0);
    su = new netscape.softupdate.SoftwareUpdate(this, "Royal Java");

    // Start the install process
    err = su.StartInstall("java/download/royalairways/RoyalJava/", vi,
        netscape.softupdate.SoftwareUpdate.LIMITED_INSTALL);
```

```
    if (err == 0) {

        // Find the Java download directory on the user's computer
        JavaFolder = su.GetFolder("Java Download");

        // Install the JAR file. Unpack it and list where it goes
        err = su.AddSubcomponent("RoyalJava JAR", vi, "rjc.jar",
            JavaFolder, "", this.force);
    }

    // Unless there was a problem, move the JAR file to its final
    // location and update the Client Version Registry
    if (err != 0)
        su.AbortInstall();
    else {
        su.FinalizeInstall();
        cAlert("Installation complete. You must restart Communicator
            to use the new Java classes.");
    }
}
```

Chapt r 5
Packaging Softwar

In this chapter:

Creating and Signing a SmartUpdate JAR File

Making the Software Available

Once you've written an installation script for your software, you need to package it for delivery. This chapter introduces the packaging process.

Creating and Signing a SmartUpdate JAR File

After you have written the software and its installation script, you must package those files and any necessary auxiliary files into a signed JAR file for delivery.

A SmartUpdate JAR file is a zip file with additional information. In general, that information may include the name of th installation script for the contents of the JAR file and may optionally include signatures for digitally signed files so that a site administrator or an end user can decide whether to install the software.

> **Important** A SmartUpdate JAR file must comply with several restrictions that do not apply to JAR archives in general. All files in the SmartUpdate JAR file *must* be signed. The JAR file *must* include an installation script that is signed by only one principal. All other files can be signed by multiple principals, but one of those principals must be the principal who signed the installation script.

Befor you can create an installable JAR archive, you must have a code-signing certificate. You use this certificat to sign your code and to identify yourself to the users of your installation script. The certificate is different from the on used in email. For details on signing files and getting a code-signing certificate, see *Netscape Object Signing*. Netscape recommends that you give your certificate a short nickname. You might have to type it in and the default names are very long.

Once you have your code-signing certificate, you use it to package your software and the installation script into an archive. You use the Netscape Signing Tool to create a signed and installable JAR archive. Be sure to use the `-i` option to specify the installation script. For details on using the signing tool, see *Signing Software with Netscape Signing Tool 1.1*.

Making the Software Available

You must publish the JAR file to make it available for content developers and end users. You may publish it publicly on the Internet or privately on your company's intranet or extranet. If your JAR file is small enough, you can even attach it to an email message.

If the software is a Netscape plug-in, you can register it with Netscape to appear in the Plug-in Finder. In addition, you can provide content developers with the URL for the trigger script so that the content developer can include the URL in pages that need the software. Providing the URL for the trigger script may be especially desirable for content developers who want to specify a particular plug-in and not just any plug-in for a particular MIME type.

If you want content providers to use a trigger script in their pages so that users can access your software, you need to give content developers a sample trigger script that covers the basics for getting the correct version of the softwar on the user's machine. Note that content developers may modify your sample script. Chapter 6, "Initiating a SmartUpdate Installation," discusses creating trigger scripts. *SmartUpdate for Content Developers* discusses what a content developer needs to about trigger scripts.

The web server on which you publish your JAR files must be configured to serve JAR files. That is, configure your HTTP server to serve JAR files with the MIME type `application/java-archive`.

If you want to register a plug-in with Netscape so that it appears in the Plug-in Find r, follow the instructions at

`http://home.netscape.com/plugins/plg_profile.html`

Last Updated: 03/11/99 11:37:16

Chapter 6
Initiating a SmartUpdate Installation

In this chapter:

- Approaches to Initiating SmartUpdate
- What's Been Added for Writing Trigger Scripts
- What Your Trigger Script Should Do
- SampleTrigger Scripts

Once an installable JAR file has been published on the Internet or on your company's intranet or extranet, end us rs and content developers can access it by initiating the SmartUpdate process in several ways. End users frequently discover they need software by having a content provider tell them so. ("To best view the cool stuff on this page, you should download the KillerApp software.") In this case, the content provider may include a script that tests to determine exactly which version of the software is appropriate for the machine and to verify whether or not it is pres nt. Alternatively, something may send the user directly to the software developer's site for the download. Whichever way it happens, the content provider or the software developer eventually provides the code that initiates SmartUpdate to get the software onto the user's machine.

This chapter discusses the ways SmartUpdate can be initiated and why you might choose each approach. It tells you how to write an appropriate JavaScript script trigger, providing information on the Java classes provided by SmartUpdate technology for this purpose, and providing sample scripts.

Approaches to Initiating SmartUpdate

Initiating SmartUpdate can be done in several ways, which you can think of as on a continuum of how much contr l you (the software developer or the content provider) exercise over what gets downloaded:

- Providing an HTML page that contains a JavaScript trigger script. That script uses LiveConnect and the `Trigger` class provided by SmartUpdate to trigger an update.

  This approach provides the most control; it is strongly recommended for reasons discussed below.

- Providing an HTML page that contains a JavaScript trigger script. That script contains a direct link to a JAR file.

  This approach provides less control; it may be appropriate, but some care is required. There are more possible end user problems with this approach; you are strongly encouraged to warn users of those problems (described in Appendix B, "End User Problems.")

- Providing a direct link to a JAR file.

  This approach is the least work for you (and the most like the present situation.) However, it provides the least c ntrol. It still may be appropriate, but you need to be very careful with this approach. If you use this approach, you are strongly encouraged to include a README file next to the SmartUpdate JAR file, to explain the circumstances under which the archive should be downloaded and to explain problems that might arise.

- Providing another vendor-supplied link (such as a vendor-supplied trigger script.)

  This approach puts all the onus on the software developer. The link provided by the developer may be to a trigger script, to another installation page, or even to the company's home page.

Table 6.1 summarizes some of advantages of using a script and `Trigger` class instead of one of the other approaches.

Tabl 6.1 Appr aches t initiating SmartUpdat

Script using `Trigger` class	Script using direct link	Direct link to JAR file
• Can query the Client Version Registry to see if this software (the same or a different version) is already installed and possibly avoid a duplicate installation	• Can't query the Client Version Registry	• Can't query the Client Version Registry
• Can check other information to ensure that SmartUpdate can proceed successfully on this machine ("Are Java and SmartUpdate enabled?") or to download a different JAR file ("What OS and language are in use?")	• Can check other information and change download strategy accordingly	• Can't check anything before download
• Consider redundantly checking some conditions in the installation script, in case the user bypasses your script	• Consider redundantly checking some conditions in the installation script, in case the user bypasses your script	• All checks must be made by the installation script, which happens after the JAR fil has been downloaded to th machine
• Can request that the installation be silent (that is, that the user not see dialog boxes during installation)	• Can't request a silent installation	• Can't request a silent installation
• Can force installation of an earlier version of the software	• Can't force installation of an earlier version	• Can't force installation of an earlier version
• JAR files are always recognized, resulting in fewer end user errors	• An improperly configured web server or proxy server may not recognize a JAR file and hence not install it properly	• An improperly configured web server or proxy server may not recognize a JAR file and hence not install it properly
• Consider including a README file in the same directory, just in case	• Consider including a README file in the same directory, just in case	• Strongly encouraged to include a README file, becaus the user must know what conditions are needed to successfully install the JAR file

The rest of this chapter discusses writing a trigger script to initiate SmartUpdate. Trigger scripts are JavaScript scripts that use Java classes to initiate SmartUpdate.

What's Been Added for Writing Trigger Scripts

To help you write a trigger script, two new Java classes have been added, `netscape.softupdate.VersionInfo` and `netscape.softupdate.Trigger`. These classes are described in Chapter 7, "Reference." Through JavaScript's LiveConnnect feature, these classes can be used with JavaScript and let you:

- Ensure that SmartUpdate is enabled on the user's machine
- Specify a version for your software
- Query the Client Version Registry
- Initiate SmartUpdate using a particular JAR file

You can, of course, also use the full range of JavaScript to implement as complex a trigger script as you need.

In addition, the JavaScript `navigator` object has sev ral prop rties that can be useful in triggering a software download. Two new properties, `language` and `platform`, may be particularly useful. For information on th `navigator` object and its properties, see the *JavaScript Guide*. For information on new JavaScript features, see *What's New in JavaScript 1.2*.

What Your Trigg r Script Should Do

The contents of your trigger script can vary in many ways. This section discusses some of the typical tasks it might perform. Not all of these tasks are necessarily appropriate for your situation; conversely, you may have special needs not discussed here.

Licensing and Registration or Payment

Frequently software on the web is licensed and may require either registration or payment. Such requirements do not change when using SmartUpdate to download and install software. Simply make sure you've gone through the appropriate steps before initiating SmartUpdate.

Decide Whether It's Possible to Use SmartUpdate

The first thing your script should do is check that Java and SmartUpdate are enabled on the user's machine. If either of these is disabled, then installation of a JAR file won't work, because the installation script inside the JAR file requires both be enabled. If either of these features is disabled, let the user know and then discontinue the script. You can make these checks with code similar to this:

```
if ( navigator.javaEnabled() ) {
   if ( netscape.softupdate.Trigger.UpdateEnabled() ) {
      // ... perform rest of trigger...
      }
   else
      alert("SmartUpdate is not enabled; enable it and try again.")
   }
else
   alert("Java is not enabled; enable it and try again.")
```

You must check that Java is enabled before you can do anything with the Java classes discussed in "What's Been Added for Writing Trigger Scripts" on page 48. You must check that SmartUpdate has been enabled before you can do anything else with those classes.

Ch ck Machine and Browser Configuration

Your script may use different JAR files depending on the circumstances of the user's particular machine. For example, you may have a different executable for different operating systems or you may have a different user interface depending on the language used in the browser. Three properties of the `navigator` object may be of particular use:

`language`	The language of the browser client software being used. Usually a two-letter code, such as en; occasionally a five-character code to indicate a language sub-type, such as zh_CN.
`platform`	The machine architecture (such as Windows 95/NT or Mac OS PowerPC) for which the running browser was compiled
`appVersion`	The version of the browser software being used.

For example, your script could use code such as the following to pick a different JAR file based on machine architecture:

```
if (navigator.platform == "SunOS5.4")
   // Trigger download for Sun
else if (navigator.platform == "Win32")
   // Trigger download for Windows NT/95
else if (navigator.platform == "MacPPC")
   // Trigger download for Mac OS PowerPC
else alert ("You've got a machine type I don't support.")
```

Check If This Software Is Already Installed

Before downloading a JAR file onto the user's machine, you may want to check to s e if the software is already there. You can thus avoid a duplicate download. If your trigger script does not check for an existing version of the software, the installation script probably will. Therefore, this is not always absolutely n cessary. However, keep in mind that the installation script can't check until after the entire JAR file has been downloaded to the machine. If you check ahead of time, you can save the end user the aggravation of waiting while the archive downloads, only to be told "Never

mind!"

For new software, SmartUpdate may be the only way it has ever been delivered. If so and if the software is on the machine, it will be registered in the Client Version Registry; you can use the SmartUpdate classes to check this. However, if the software package has previously been available without the benefit of a SmartUpdate installation, you cannot rely on it being in the Client Version Registry. In this situation, you must also do some checking specific to your software.

To check the Client Version Registry for an installed version and decide whether to trigger SmartUpdate on that basis, use code similar to the following:

```
vi = new netscape.softupdate.VersionInfo(2,0,1,0);
trigger = netscape.softupdate.Trigger;

trigger.ConditionalSoftwareUpdate(
   "http://royalairways/royalpkg.jar",
   "plugins/royalairways/RoyalPI", vi, trigger.DEFAULT_MODE);
```

The `ConditionalSoftwareUpdate` method of the `Trigger` object takes 4 arguments:

- The location of the JAR file to download
- The name used to identify the software in the Client Version Registry (for information on how you pick a name for the registry, see "Positioning Software in the Client Version Registry" on page 33)
- The version of the software being downloaded
- A flag determining whether to download quietly (described in "Installing Silently" on page 54)

For details on this method, see its description in Chapter 7, "Reference." In this example, SmartUpdate occurs only if the Client Version Registry does not contain version 2.0.1.0 or later of the `plugins/royalairways/RoyalPI` package.

If the software has previously been available without a SmartUpdate installation, it may or may not appear in the Client Version Registry. (It will appear if this version has already been installed on the machine; it won't if an earlier version was installed.) To update or install when an earlier version of the software could have been installed without SmartUpdate, you need to perform these steps:

1. Test to see if the MIME type associated with the software package is already registered with Communicator.

   If not, the software hasn't been installed on this machine and the script can proceed with the normal SmartUpdate installation. If the MIME type is registered, then a previous version of the software may exist on the machine.

2. Next, check to see if there is version information in the Client Version Registry for the software.

   If there is, the software was previously installed using SmartUpdate and SmartUpdate and so the trigger classes can check the versions.

3. If there is no version information in the Client Version Registry, the software was previously installed without using SmartUpdate.

   In this case, the script must have application-specific code to test the installed version against the current one and then proceed with installation if necessary.

To test these conditions, you can use code similar to the following

```
function startDownload(minVersion) {
   var myMimetype = navigator.mimeTypes["application/royalmime"];
   var trigger = netscape.softupdate.Trigger;

   // If some version is already installed on this machine...
   if ( myMimetype )   {

      // Get the SmartUpdate version information
      version = trigger.GetVersionInfo("/royalairways/royalsw");

      // Installed by SmartUpdate, let it take care of version checking
```

```
    if (version != null)   {
       return trigger.ConditionalSoftwareUpdate(
          "http://royalairways/royalpkg.jar",
          "/royalairways/royalsw",
          new netscape.softupdate.VersionInfo(minVersion, 0, 0, 0),
          trigger.DEFAULT_MODE);
    }

    // No SmartUpdate information, so it was installed some other way.
    else     {
       // Put plug-in specific code for checking the version here.
    }
  }

  // No version of RoyalSW is currently installed on this machine,
  // so start the download
  else
     return trigger.StartSoftwareUpdate(
        "http://royalairways/royalpkg.jar", trigger.DEFAULT_MODE);

  return false;
}

startDownload (0); //Install any version.
</SCRIPT>
```

Downgrading a Package

In some circumstances you may want to trigger the update even if a version of the software exists with a higher version number. For example, you may want to revert to an earlier version of a software package if a later version does not suit your needs. You can accomplish this by using the `StartSoftwareUpdate` method instead of the `ConditionalSoftwareUpdate` method and passing the `FORCE_MODE` flag in its last parameter. This flag requests that an installation be allowed to override a more recent version of a component. The `StartSoftwareUpdate` method takes two parameters:

- The location of the JAR file to download
- A flag determining whether to download silently and whether to force a downgrade.

Refer to its description in the reference for details of this method.

The following script fragment unconditionally triggers installation of version 2.0.1 of the RoyalSW software:

```
vi = new netscape.softupdate.VersionInfo(2,0,1,0);
trigger = netscape.softupdate.Trigger;

trigger.StartSoftwareUpdate (
   "http://royalairways/royalpkg.jar", trigger.FORCE_MODE);
```

In the corresponding installation script, for the request to be complied with all calls to the `SoftwareUpdate` object's `AddSubcomponent` method must pass `this.force` as their last argument. The `this.force` argument reflects the value of the `FORCE_MODE` flag. For information on writing installation scripts, see Chapter 4, "Writing an Installation Script."

If you need to use both the `FORCE_MODE` flag and the `SILENT_MODE` flag described below, the call to `StartSoftwareUpdate` would look as follows:

```
trigger.StartSoftwareUpdate (
   "http://royalairways/royalpkg.jar",
   trigger.FORCE_MODE | trigger.SILENT_MODE);
```

Installing Silently

A system administrator using the AutoUpdate feature of Mission Control to automatically upgrade software on their users' machines may wish the upgrade to happen quietly, without displaying dialog boxes or asking questions of the user. You can accomplish this by passing the `SILENT_MODE` flag in the last parameter to the `ConditionalSoftwareUpdate` or `StartSoftwareUpdate` method.

If this flag is included, the progress and permission dialog boxes may not appear while the software is being downloaded and installed. However, this flag does not suppress the appearance of security dialog boxes.

A silent installation can occur only if the signer of the JAR file has the `SilentInstall` privilege. This privilege can be set only using Netscape Mission Control.

The following script fragment silently triggers installation of version 2.0.1 of the RoyalSW software, if necessary:

```
vi = new netscape.softupdate.VersionInfo(2,0,1,0);
trigger = netscape.softupdate.Trigger;

trigger.ConditionalSoftwareUpdate (
   "http://royalairways/royalpkg.jar","/royalairways/RoyalSW", vi,
   trigger.SILENT_MODE);
```

For the request to be complied with, the corresponding installation script must be written accordingly. The `SILENT_MODE` flag is indicated in the installation script by the value of `this.silent`. If `this.silent` is true, the installation script should suppress the display of dialog boxes.

If you need to use both the `SILENT_MODE` flag and the `FORCE_MODE` flag described in "Downgrading a Package" on page 53, the call to `StartSoftwareUpdate` would look as follows:

```
trigger.StartSoftwareUpdate (
   "http://royalairways/royalpkg.jar",
   trigger.FORCE_MODE | trigger.SILENT_MODE);
```

SampleTrigger Scripts

The rest of this chapter gives simple examples of trigger scripts that illustrate some of the possibilities.

Simplest Trigger Script

T initiate an update from JavaScript, the only requirements are that Java and SmartUpdate be enabled for the browser and that you have the URL from which to download the JAR file. The following sample code uses the `StartSoftwareUpdate` method to unconditionally trigger a download from `http://royalairways/royalpkg.jar`.

```
if ( navigator.javaEnabled() ) {

   trigger = netscape.softupdate.Trigger;
   if ( trigger.UpdateEnabled() )
      trigger.StartSoftwareUpdate (
         "http://royalairways/royalpkg.jar", trigger.DEFAULT_MODE);
   else
      document.write("Enable SmartUpdate before running this script.");
   }

else
      document.write("Enable Java before running this script.");
```

Checking Machine Architecture and Language

This example does the following:

- Checks to make sure Java and SmartUpdate are enabled
- Creates a version info object for version 4.0.1.0
- Checks the machine architectur (Windows 95/98/NT or Solaris 5.5) and user int rface language (Japanese or English)
- Downloads one of 4 JAR files based on these settings

The script follows:

```
if (navigator.javaEnabled() ) {

    trigger = netscape.softupdate.Trigger;
    if (trigger.UpdateEnabled() ) {
       vi = new netscape.softupdate.VersionInfo(4,0,1,0);

       // Downloads for Windows 95/NT
       if (navigator.platform == "Win32") {
          if (navigator.language == "jp") // Japanese
             trigger.ConditionalSoftwareUpdate (
                "http://www.royalairways.com/RoyalWin32JP.jar",
                "plugins/royalairways/RoyalSW", vi,
                trigger.DEFAULT_MODE);

          else if (navigator.language == "en") // English
             trigger.ConditionalSoftwareUpdate (
                "http://www.royalairways.com/RoyalWin32EN.jar",
                "plugins/royalairways/RoyalSW", vi,
                trigger.DEFAULT_MODE);
       }

       // Download for Solaris
       else if (navigator.platform == "SunOS5.5") {
          if (navigator.language == "jp")
             trigger.ConditionalSoftwareUpdate (
                "http://www.royalairways.com/RoyalSunJP.jar",
                "plugins/royalairways/RoyalSW", vi,
                trigger.DEFAULT_MODE);
          else if (navigator.language == "en")
             trigger.ConditionalSoftwareUpdate (
                "http://www.royalairways.com/RoyalSunEN.jar",
                "plugins/royalairways/RoyalSW", vi,
                trigger.DEFAULT_MODE);       }     } }
```

Incremental Updates

If you provide an incremental update (for example, from version 3.0 to version 3.0.2 of your software), you may want to check the version number of the previously installed software and update only some of the components based on the result. You might do this, for example, if your software is very large, but the update affects only a small number of files. By having separate JAR files for complete and incremental updates, you can significantly reduce downl ad times.

This sample covers these cases:

- If the software either isn't installed or a version earlier than 3.0 is installed, trigger a complete installation.
- If version 3.0.0 or 3.0.1 is installed, incrementally upgrade to version 3.0.2

The script follows:

```
// Creates version objects for versions 3.0.0, 3.0.1 and 3.0.2
v3_0 = new netscape.softupdate.VersionInfo (3,0,0,35);
v3_1 = new netscape.softupdate.VersionInfo (3,0,1,25);
v3_2 = new netscape.softupdate.VersionInfo (3,0,2,50);

// Get the version number for the currently installed software
installed_version = netscape.softupdate.Trigger.GetVersionInfo(
   "plugins/royalairways/RoyalSW");

// If the installed version is null (that is, not installed)
// or less than version 3.0, do a complete install.
if (installed_version == null ||
    installed_version.compareTo(v3_0) < 0 )
      StartSoftwareUpdate("http://www.royalairways.com/v3_2.jar",
         trigger.DEFAULT_MODE);

// If the installed version is 3.0, do this update
```

```
else if (installed_version.compareTo(v3_1) < 0)
   netscape.softupdate.Trigger.ConditionalSoftwareUpdate (
      "http://www.royalairways.com/incremental_v0_to_v2.jar",
      "plugins/royalairways/RoyalSW", v3_2, trigger.DEFAULT_MODE);

// If the installed version is 3.1, do this update
else if (installed_version.compareTo(v3_2) < 0)
   netscape.softupdate.Trigger.ConditionalSoftwareUpdate (
      "http://www.royalairways.com/incremental_v1_to_v2.jar",
      "plugins/royalairways/RoyalSW", v3_2, trigger.DEFAULT_MODE);
```

Chapter 7
R fer nce

This chapter provides reference material for the Java objects that can be used with JavaScript to support SmartUpdate trigger scripts.

All of the objects are in the `netscape.softupdate` package. The objects are

- `SoftwareUpdate` object
- `Trigger` object
- `VersionInfo` object
- `WinProfile` object
- `WinReg` object

For general information on JavaScript, see the *JavaScript Guide*. For general information on writing plug-ins, see th *Plug-in Guide*.

SoftwareUpdate

You use this object to manage the downloading and installation of software with the JAR Installation Manager. This object and its methods are used in installation scripts.

In Package

`netscape.softupdate`

Meth d Summary

`SoftwareUpdate`	Creates a `SoftwareUpdate` object.
`AbortInstall`	Aborts the downloading and installation of the software.
`AddDirectory`	Unpacks an entire subdirectory.
`AddSubcomponent`	Unpacks a single file.
`DeleteComponent`	Deletes the specified file and its entry in the Client Version Registry.
`DeleteFile`	Deletes the specified file.
`DiskSpaceAvailable`	Determines whether the specified amount of disk space is available.
`Execute`	Extracts a file from the JAR file to a temporary location and schedules it for later execution.
`FinalizeInstall`	Finalizes the installation of the software.
`Gestalt`	Retrieves information about the operating environment. (Mac OS only)
`GetComponentFolder`	Returns an object representing the directory in which a component is installed.
`GetFolder`	Returns an object representing a directory, for use with the `AddSubcomponent` method.
`GetLastError`	Returns the most recent non-zero error code.
`GetWinProfile`	Constructs an object for working with a Windows .ini file.
`GetWinRegistry`	Constructs an object for working with the Windows Registry.
`Patch`	Applies a set of differences between two versions.
`ResetError`	Resets a saved error code to zero.
`SetPackageFolder`	Sets the default package folder that is saved with the root node.
`StartInstall`	Initializes installation for the given software and version.
`Uninstall`	Removes a package that was previous installed by SmartUpdate.

SoftwareUpdate

Creates a `SoftwareUpdate` object.

Syntax

```
SoftwareUpdate (
   JSObject env,
   String inUserPackageName);
```

Parameters

The `SoftwareUpdate` constructor takes the following parameters:

`env` An object that provides the JavaScript context to the constructor. Always use `this` for this parameter.

`inUserPackageName` A string used in user prompts to name this software.

Returns

A new `SoftwareUpdate` object.

Description

You must call the `StartInstall` method after you call this constructor. It is an error to call any other `SoftwareUpdate` methods before calling `StartInstall`.

Example

Use the following code to create a `SoftwareUpdate` object and use the string "Royal Airways Software" in dialog boxes:

```
su = new netscape.softupdate.SoftwareUpdate(this, "Royal Airways Software");
```

AbortInstall

Aborts installation of the software; performs cleanup of temporary files.

Meth d of

`SoftwareUpdate`

Syntax

```
int AbortInstall();
```

Parameters

Non .

Returns

An integer error code. For a list of possible values, see "Return Codes" on page 102.

Example

Use the following code to abort or to finalize an installation, based on a variable you set earlier in your code:

```
su = new netscape.softupdate.SoftwareUpdate(this, "Royal Airways Software");

...

if (bAbort)
   su.AbortInstall();
else {
   err = su.FinalizeInstall();
```

AddDirectory

Unpacks an entire dir ctory into a temporary location and enters each file in the directory into the Client Version Registry.

Method f

SoftwareUpdate

Syntax

```
public int AddDirectory (
   String registryName,
   VersionInfo version,
   String jarSourcePath,
   Object localDirSpec,
   String relativeLocalPath,
   Boolean forceUpdate); (Communicator 4.05 or later)

public int AddDirectory (
   String jarSourcePath); (Communicator 4.5 or later)

public int AddDirectory (
   String registryName,
   String jarSourcePath,
   Object localDirSpec,
   String relativeLocalPath); (Communicator 4.5 or later)

public int AddDirectory (
   String registryName,
   String version,
   String jarSourcePath,
   Object localDirSpec,
   String relativeLocalPath); (Communicator 4.5 or later)

public int AddDirectory (
   String registryName,
   String version,
   String jarSourcePath,
   Object localDirSpec,
   String relativeLocalPath,
   Boolean forceUpdate); (Communicator 4.5 or later)
```

Parameters

The `AddDirectory` method has the following parameters:

`registryName` The pathname in the Client Version Registry for the root directory of the files that are to be installed.

This parameter can be an absolute pathname (beginning with a /) or a relative pathname, (not beginning with a slash).

An absolute pathname is used as specified. A relative pathname is appended to the registry name of the package as specified by the `package` parameter to the `StartInstall` method.

If you want the pathname to be relative to the current Communicator installation instead of to the package being installed, use the prefix "`=COMM=/`" as the beginning of the pathname.

If you want the pathname to be relative to the current user folder, use the prefix "`=USER=/`".

This parameter can also be null, in which case the `jarSourcePath` parameter is used as a relative pathname.

Note that the registry pathname is not the location of the software on the computer; it is the location of information about the software inside the Client Version Registry. For information on choosing an appropriate registry name, see "Positioning Software in the Client Version Registry" on page 33.

`version` A `VersionInfo` object containing the version number for all installed files. This parameter can be null, in which case the package version from `StartInstall` is used.

`jarSourcePath` A string specifying the location of the directory within the JAR file.

An empty string ("") causes the creation of a subdirectory node in the registry without actually unpacking any files, which may be useful when you are updating a package that contains subcomponents that could also be updated separately. When `jarSourcePath` is an empty string, `registryName` cannot be null.

`localDirSpec` An object representing a directory. The directory is installed under this directory on the user's computer. You create this object by passing a string representing the directory to the `GetFolder` method.

`subdir` The name of a directory to append to `localDirSpec`, using '/' as the path separator regardless of the platform.

If `subdir` is missing, `null`, or "", the filenames are appended directly to the folder name specified by `localDirSpec`.

`forceUpdate` If `true`, replaces an existing file regardless of its version. If `false`, replaces an existing file only if the replacement has a higher version number. If `forceUpdate` is missing, `this.force` is assumed.

In most cases, you should use `this.force` as the value for this parameter. This value reflects the flag passed to the `ConditionalSoftwareUpdate` or `StartSoftwareUpdate` method of the `Trigger` object. For shared system DLLs, you may want to explicitly use `false`, so that shared components are not downgraded for other applications. For a fuller discussion, see "Downgrade a Component If Requested" on page 31.

Returns

An integer error code. For a list of possible values, see "Return Codes" on page 102. In some situations, `AddDirectory` may return other errors. For example, if the error was with regard to the signing of the JAR file, `AddDirectory` returns a security error code.

Description

The `AddDirectory` method puts the files in the specified directory in a temporary location. To move the files and all other subcomponents to their final location, call the `FinalizeInstall` method after you've successfully added all subcomponents.

The files are installed if one of the following conditions is true:

- There is no entry for a file in the Client Version Registry
- There is an entry in the Client Version Registry *but*

- the corresponding file has been deleted, or
- the version information of the `version` parameter is higher than that of the current entry in the registry, or
- the version information of the `version` parameter or of the current entry in the registry is null

Windows only: Some Windows executables and DLLs have embedded `FILEVERSION` information in the version info block of the file resource. If this information is present in *both* the already installed version of the component and the component to be installed now, then the information in the two files is compared. If the `FILEVERSION` information in the prospective new component is equal to or greater than that of the existing component, the new component is installed. Otherwise, it is not. This check is not made if either component doesn't contain `FILEVERSION` information.

Consequently, if your Windows plug-in contains `FILEVERSION` information in the header of its version info block, be sure to increment this value for each release. Note that this information is four comma-separated integers in the header of the version information block; it is not the `FileVersion` string in the body of the version information block.

AddSubcomponent

Unpacks a single subcomponent into a temporary location. Queues the subcomponent for addition to the Client Version Registry and installation to its final destination.

Method of

`SoftwareUpdate`

Syntax

```
public int AddSubcomponent (
   String registryName,
   VersionInfo version,
   String jarSourcePath,
   Object localDirSpec,
   String relativeLocalPath,
   Boolean forceUpdate); (Communicator 4.0 or later)

public int AddSubcomponent (
   String registryName,
   String version,
   String jarSourcePath,
   Object localDirSpec,
   String relativeLocalPath,
   Boolean forceUpdate); (Communicator 4.5 or later)

public int AddSubcomponent (
   String jarSourcePath); (Communicator 4.5 or later)

public int AddSubcomponent (
   String registryName,
   String jarSourcePath,
   Object localDirSpec,
   String relativeLocalPath); (Communicator 4.5 or later)

public int AddSubcomponent (
   String registryName,
   String version,
   String jarSourcePath,
   Object localDirSpec,
   String relativeLocalPath); (Communicator 4.5 or later)
```

Parameters

The `AddSubcomponent` method has the following parameters:

`registryName`	The pathname in the Client Version Registry about the subcomponent. This parameter can be an absolute pathname, such as `/royalairways/RoyalSW/executable` or a relative pathname, such as `executable`. Typically, absolute pathnames are *only* used for shared components, or components that come from another vendor, such as `/Microsoft/shared/msvcrt40.dll`. Typically, relative pathnames are relative to the main pathname specified in the `StartInstall` method. If you want the pathname to be relative to the current Communicator installation instead of to the package being installed, use the prefix `"=COMM=/"` as the beginning of the pathname. If you want the pathname to be relative to the current user folder, use the prefix `"=USER=/"`. This parameter can also be null, in which case the `jarSourcePath` parameter is used as a relative pathname. Note that the registry pathname is not the location of the software on the machine; it is the location of information about the software inside the Client Version Registry. For information on choosing an appropriate registry name, see "Positioning Software in the Client Version Registry" on page 33.
`version`	A `VersionInfo` object (Communicator 4.0) or a `String` (Communicator 4.5) containing the version number for the subcomponent. If a string, the format of `version` is "4.0.1.1234". This parameter can be null. In this case, no version is associated with this component and it is always updated.
`jarSourcePath`	A string specifying the location of the subcomponent within the JAR file.
`localDirSpec`	An object representing a directory. The subcomponent is installed under this directory on the user's machine. You create this object by passing a string representing the directory to the `GetFolder` method.
`relativeLocalPath`	A pathname relative to the `localDirSpec` parameter. The subcomponent is installed in this location on the user's machine. You must always use forward slashes (*/*) in this pathname, regardless of the convention of the underlying operating system. If this parameter is blank or `NULL`, `jarSourcePath` is used.
`forceUpdate`	If `true`, replaces an existing component regardless of its version. If `false`, replaces an existing component only if the replacement has a higher version number. In most cases, you should use `this.force` as the value for this parameter. This value reflects the flag passed to the `ConditionalSoftwareUpdate` or `StartSoftwareUpdate` method of the `Trigger` object. For shared system DLLs, you may want to explicitly use `false`, so that shared components are not downgraded for other applications. For a full discussion, see "Downgrade a Component If Requested" on page 31.

Returns

An integer error code. For a list of possible values, see "Return Codes" on page 102. In some situations the method may return other errors. For example, if the error was with regard to the signing of the JAR file, `AddSubcomponent` returns a security error.

Description

The `AddSubcomponent` method puts the subcomponent in a temporary location. To move this and all other subcomponents to their final location, call the `FinalizeInstall` method after you've successfully added all subcomponents.

The component is installed if one of the following conditions is true:

- There is no entry for this file in the Client Version Registry
- There is an entry in the Client Version Registry *but*
  - the actual file has been deleted, or
  - the version information of the `version` parameter is higher than that of the current entry in the registry, or

- the version information of the `version` parameter or of the current entry in the registry is null

Windows only: Some Windows executables and DLLs have embedded `FILEVERSION` information in the version info block of the file resource. If this information is present in *both* the already installed version of the component and the component to be installed now, then the information in the two files is compared. If the `FILEVERSION` information in the prospective new component is equal to or greater than that of the existing component, the new component is installed. Otherwise, it is not. This check is not made if either component doesn't contain `FILEVERSION` information.

Consequently, if your Windows plug-in contains `FILEVERSION` information in the header of its version info block, be sure to increment this value for each release. Note that this information is four comma-separated integers in the header of the version information block; it is not the `FileVersion` string in the body of the version information block.

Examples

The following examples show different uses of the `registryName` parameter:

```
su.StartInstall("plugins/RoyalPlug",version,
   netscape.softupdate.SoftwareUpdate.FULL_INSTALL);

// To create node /netscape/Communicator #?/plugins/royalplug/npplug
su.AddSubcomponent("npplug", ...);

// To create node /MS/Shared/Ctl3d.dll
su.AddSubcomponent("/MS/Shared/ctl3d.dll", ...);

// To create node /netscape/Communicator #?/NetHelp/royalplug/royalhelp.html
su.AddSubcomponent("=COMM=/NetHelp/royalplug/royalhelp.html",...);
```

D l teComponent

Deletes the specified file and removes its entry from the Client Version Registry.

Meth d of

`SoftwareUpdate`

Syntax

```
int DeleteComponent
   (String registryName);  (Communicator 4.5 or later)
```

Parameters

The `DeleteComponent` method has the following parameter:
`registryName` The pathname in the Client Version Registry for the file that is to be deleted.

Returns

An integer error code. For a list of possible values, see "Return Codes" on page 102.

Description

The `DeleteComponent` method deletes the specified file and removes the file's entry from the Client Version Registry. If the file is currently being used, the name of the file that is to be deleted is saved and Communicator attempts to delete it each time it starts up until the file is successfully deleted. This method is used to delete files that cannot be removed by the `Uninstall` method or to remove files that are no longer necessary or whose names have changed.

D let File

Deletes the specified fil but does not r move it from the Cli nt V rsion Registry.

Method of

SoftwareUpdate

Syntax

```
int DeleteFile
    (Object folder,
    String relativeFileName); (Communicator 4.5 or later)
```

Parameters

Th DeleteFile method has the following parameters:

localDirSpec An object representing the directory from which the file is to be deleted.

relativeLocalPath A pathname relative to localDirSpec representing the file that is to be deleted.

Returns

An integer error code. For a list of possible values, see "Return Codes" on page 102.

Description

The DeleteFile method deletes the specified file but does not remove the file's entry from the Client Version Registry. If the file is currently being used, the name of the file that is to be deleted is saved and Communicator attempts to delete it each time it starts up until the file is successfully deleted. This method is used to delete files that w re not installed or created by a SmartUpdate process.

DiskSpaceAvailable

Gets the amount of space that is available on a disk.

Method of

SoftwareUpdate

Syntax

```
long DiskSpaceAvailable (
   Object localDirSpec); (Communicator 4.5 or later)
```

Parameters

The DiskSpaceAvailable method has the following parameter:

localDirSpec An object representing a directory obtained by GetFolder.

Returns

The number of bytes available on the drive that contains localDirSpec.

Description

The DiskSpaceAvailable method gets the amount of disk space that is available on the drive that contains localDirSpec. Other processes may be using disk space, or your installation process may require more space than th installed files, so a a return value that is equal to the size of your installed files is not a guarantee that there is nough space to perform the installation.

Execute

Copies an ex cutable fil from a JAR file to a temporary location and schedules it for later xecuti n.

M th d f

SoftwareUpdate

Syntax

```
int Execute (
   jarSourcePath)  (Communicator 4.0 or later)

int Execute (
   String jarSourcePath,
   String args);  (Communicator 4.5 or later)
```

Parameters

The `Execute` method has the following parameters:

`jarSourcePath` The pathname of the file to extract and execute.

`args` A parameter string that is passed to the executable. (Ignored on Mac OS)

Returns

An integer error code. For a list of possible values, see "Return Codes" on page 102.

Description

The `Execute` method extracts the named file from the JAR file to a temporary file name. Your code must call the `FinalizeInstall` method to actually execute the file. You could use this method to launch an InstallShield installer stored in a JAR file.

Exampl

The following example performs an installation by retrieving and running an executable file:

```
version = new netscape.softupdate.VersionInfo(2, 1, 0, 0);
su = new netscape.softupdate.SoftwareUpdate(this, "Royal Software");
su.StartInstall("plugins/RoyalSW", version,
   netscape.softupdate.SoftwareUpdate.FULL_INSTALL);
su.Execute("royalpkg.exe");
su.FinalizeInstall();
```

FinalizeInstall

Finalizes the installation of the software. Moves all components to their final locations, launches any pending executions, and registers the package and all of its subcomponents in the Client Version Registry.

Method of

`SoftwareUpdate`

Syntax

```
int FinalizeInstall();
```

Parameters

None.

Returns

An integer error code. For a list of possible values, see "Return Codes" on page 102. In some situations the method may return other errors. For example, if the error was with regard to the signing of the JAR file, it returns a s curity error. In a few cases you may get a registry error.

Example

Use the following code to abort or to finalize an installation, based on a variable you set earli r in your code:

```
su = new netscape.softupdate.SoftwareUpdate(this, "Royal Airways Software");

if (bAbort)
```

```
   su.AbortInstall();
else {
   err = su.FinalizeInstall();
```

Gestalt

(Macintosh only)

Retrieves information about the operating environment.

Meth d of

`SoftwareUpdate`

Syntax

```
OSErr Gestalt (
   String selector,
   long * response);
```

Parameters

The `Gestalt` method takes the following parameters:

`selector` The selector code for the information you want.

`response` On return, the requested information. The format depends on the select code specified in the `selector` parameter.

Description

The `Gestalt` method is a wrapper for the `Gestalt` function of the Macintosh Toolbox. For information on that function, see *Inside Macintosh: Operating System Utilities.*

This method returns null on Unix and Windows platforms.

G tComponentFolder

Returns an object representing the directory in which a component is installed.

Meth d of

`SoftwareUpdate`

Syntax

```
Object GetComponentFolder
   (String registryName)  (Communicator 4.0 or later)

Object GetComponentFolder (
   String registryName,
   String subDirectory);  (Communicator 4.5 or later)
```

Parameters

The `GetComponentFolder` method has these parameters:

`registryName` The pathname in the Client Version Registry for the component whose installation directory is to be obtained.

`subDirectory` A string that specifies the name of a subdirectory. If the specified subdirectory doesn't exist, it is created. This parameter is available in Communicator 4.5 or later and may be case sensitive (depending on the operating system).

Returns

An object representing the dir ctory in which the component is installed, or `NULL` if the component could not be found or if `subDirectory` refers to a file that already exists.

Description

The `GetComponentFolder` method to find the location of a previously installed software package. Typically, you use this method with the `AddSubcomponent` method or the `AddDirectory` method.

Example

Use the following code to find the fold r containing th Royal Airways plug-in:

```
su = new netscape.softupdate.SoftwareUpdate(this, "Royal Software");
folder = su.GetComponentFolder("plugins/RoyalSW");
```

GetFolder

Returns an object representing one of Netscape's standard directories.

Method f

`SoftwareUpdate`

Syntax

```
Object GetFolder (
   String FolderName);  (Communicator 4.0 or later)

Object GetFolder (
   String folderName,
   String subDirectory);  (Communicator 4.5 or later)

Object GetFolder (
   Object localDirSpec,
   String subDirectory);  (Communicator 4.5 or later)
```

Parameters

The `GetFolder` method has the following parameters:

`folderName` A string representing one of Netscape's standard directories. There are two sets of possible values for this parameter. The first set contains platform-independent locations; the second set contains platform-specific locations. You are encouraged to use the platform-independent locations. S the list in the Description section for the two sets of locations. In Communicator 4.5 `foldername` is not case sensitive; in Communicator 4.0, `foldername` must match a exactly.

`subDirectory` A string that specifies the name of a subdirectory. If the specified subdirectory doesn't exist, it is created. This parameter is available in Communicator 4.5 or later and may be case sensitive (depending on the operating system).

`localDirSpec` An object representing a directory obtained by `GetComponentFolder` or `GetFolder`. This parameter is available in Communicator 4.5 or later.

Returns

An object representing one of Netscape's standard directories, or `NULL` in case of error or if `subDirectory` refers to a file that already exists.

Description

The `GetFolder` method obtains an object representing one of Netscape's standard directories, for use with the `AddSubcomponent` and `GetWinProfile` methods.

The value of `folderName` must be one of the following:

Platform-independent locations	Platform-dependent locations
"Communicator"	"Mac System"
"Current User"	"Mac Desktop"
"Java Download"	"Mac Trash"
"Plugins"	"Mac Startup"
"Program"	"Mac Shutdown"
"Netscape Java Bin"	"Mac Apple Menu"
"Netscape Java Classes"	"Mac Control Panel"
"Temporary"	"Mac Extension"
"NetH lp" (Communicator 4.5 or later)	"Mac Fonts"
"OS Drive" (Communicator 4.5 or later)	"Mac Preferences"
"file:///" (Communicator 4.5 or later)	"Unix Local"
"User Pick"	"Unix Lib"
	"Win System"
	"Win System16"
	"Windows"

The value "User Pick" is a special case. If you specify this directory, the end user is presented with a dialog box and asked to choose the directory to use. With versions of prior to Communicator 4.5, the dialog box appears onc and causes Communicator to set the directory the user chooses as a default that is used by subsequent installs.

In Communicator 4.5 or later, when "User Pick" is specified, the end user is always presented with a dialog box and asked to choose the directory to use.

The "file:///" form is only valid when the `subDirectory` parameter is used. It must be in file: URL format minus the "file:///" part. For example,

```
mydir = su.GetFolder("file:///", "cl/mysoftco/somedir);
```

Note that forward slashes are used, regardless of the platform.

The folders whose names start with "Win", "Mac", or "Unix" are specific to those platforms. You should be careful about using one of those directories, as it makes your installation platform-specific.

Example

To get an object representing the standard plug-ins directory, you would use this call:

```
plugindir = su.GetFolder("Plugins");
```

GetLastError

Returns the most recent nonzero error code.

Method of

`SoftwareUpdate`

Syntax

`int GetLastError ();` *(Communicator 4.5 or later)*

Parameters

None.

Returns

The most recent nonz ro error cod . For a list of possible valu s, see "Return Codes" on page 102.

D scription

The `GetLastError` method to obtain the most recent nonzero rror code since `StartInstall` or `ResetError` were called. This method allows you defer checking for rror codes each time you call `Addsubcomponent` or

AddDirectory until the last AddSubcomponent or AddDirectory call.

The GetLastError method does not return errors from methods that return objects, such as GetFolder.

Example

The following example calls GetLastError after a series of AddSubcomponent calls:

```
su.AddSubcomponent("npplug", ...);

su.AddSubcomponent("/MS/Shared/ctl3d.dll", ...);

su.AddSubcomponent("=COMM=/NetHelp/royalplug/royalhelp.html",...);

err = su.GetLastError();
```

GetWinProfile

(Windows only)

Constructs an object for working with a Windows .ini file.

Method f

SoftwareUpdate

Syntax

```
WinProfile GetWinProfile (
   Object folder,
   String file);
```

Parameters

The GetWinProfile method has the following parameters:

folder An object representing a directory. You create this object by passing a string representing the directory to the GetFolder method.

file A relative pathname to an initialization file in the directory specified by the folder parameter, such as "royal/royal.ini".

Returns

A WinProfile object.

Description

The GetWinProfile method creates an object for manipulating the contents of a Windows .ini file. Once you have this object, you can call its methods to retrieve strings from the file or to add strings to the file. For information on the returned object, see WinProfile.

This method returns null on Unix and Macintosh platforms.

Example

To edit the win.ini file, you would create a WinProfile object with this call:

```
su.GetWinProfile (su.GetFolder("Windows"), "win.ini");
```

GetWinRegistry

(Windows only)

Constructs an object for working with the Windows Registry.

Meth d f

SoftwareUpdate

Syntax

```
WinReg GetWinRegistry();
```

Parameters

None.

Returns

A WinReg object.

Description

Th GetWinRegistry method to create an object for manipulating the contents of the Windows Registry. Once you have this object, you can call its methods to retrieve or change the registry's content. For information on the returned object, see WinReg.

This method returns null on Unix and Macintosh platforms.

Example

To use the HKEY_USERS section of the Windows registry, use these statements:

```
su = new netscape.softupdate.SoftwareUpdate(this, "Royal Software");
winreg = su.GetWinRegistry();
winreg.SetRootKey(winreg.HKEY_USERS);
```

Patch

Updates an existing component.

Method of

SoftwareUpdate

Syntax

```
int Patch (
   String registryName,
   String jarSourcePath,
   Object localDirSpec,
   String relativeLocalPath);  (Communicator 4.5 or later)

int Patch (
   String registryName,
   VersionInfo version,
   String jarSourcePath,
   Object localDirSpec,
   String relativeLocalPath);  (Communicator 4.5 or later)

int Patch (
   String registryName,
   String version,
   String jarSourcePath,
   Object localDirSpec,
   String relativeLocalPath);  (Communicator 4.5 or later)
```

Parameters

The Patch m thod has the following parameters:

registryName	The pathname in the Client Version Registry for the component that is to be patched. This parameter can be an absolute pathname, such as `/royalairways/RoyalSW/executable` or a relative pathname, such as `executable`. Typically, absolute pathnames are *only* used for shared compon nts, or components that come from another vendor, such as `/Microsoft/shared/msvcrt40.dll`. Typically, relative pathnames are relative to the main pathname specified in the `StartInstall` method. If you want the pathname to be relative to the current Communicator installation instead of to the package being installed, use the prefix `":=COMM=/"` as the beginning of the pathname. If you want the pathname to be relative to the current user folder, use the prefix `":=USER=/"`. This parameter can also be null, in which case the `jarSourcePath` parameter is used as a relative pathname. Note that the registry pathname is not the location of the software on the computer; it is the location of information about the software inside the Client Version Registry. For information on choosing an appropriate registry name, see "Positioning Software in the Client Version Registry" on page 33.
version	An `VersionInfo` object containing the version number for the subcomponent. This parameter can be null. In this case, no version is associated with this component and it is always updated.
jarSourcePath	A string specifying the location of the differences file within the JAR file. To create the differences file, use the `NSDiff` utility, which is described in Appendix D, "The NSDiff Utility."
localDirSpec	An object representing the directory in which the subcomponent that is to be patched resides. You create this object by passing a string representing the directory to the `GetFolder` method.
relativeLocalPath	A pathname relative to the `localDirSpec` parameter that identifies the subcomponent that is to be patched. You must always use forward slashes (/) in this pathname, regardless of the convention of the underlying operating system. If this parameter is blank or `NULL`, `jarSourcePath` is used.

Returns

An integer error code. For a list of possible values, see "Return Codes" on page 102.

Description

The `Patch` method to update an existing component by applying a set of differences between two known versions. The set of differences is in `GDIFF` format and is created by the `NSDiff` utility. For information about using `NSDiff`, see Appendix D, "The NSDiff Utility."

A patch can only be applied between two known versions. If the existing version of the file does not match the checksum stored in the `GDIFF` file, `Patch` returns an error without applying the patch. After `Patch` applies a patch, it compares a checksum.of the resulting file against a checksum stored in the `GDIFF` file. If the checksums do not match, the original version of the file is preserved, the patched version of the file is discarded, and an error code is returned.

Any single installation process can apply multiple patches to the same file.

If `FinalizeInstall` indicates that a reboot is necessary to complete the installation, `Patch` may not work in subsequent SmartUpdate processes until the reboot is performed.

ResetError

Resets a saved error code to z ro.

Method f

`SoftwareUpdate`

Syntax

```
void ResetError (); (Communicator 4.5 or later)
```

Parameters

None.

Returns

Nothing.

Description

The `ResetError` method resets any saved error code to zero. See `GetLastError` for additional information.

Example

To reset the last error code to zero:

```
su.ResetError();
```

SetPackageFolder

Sets the default package folder.

M thod of

`SoftwareUpdate`

Syntax

```
void SetPackageFolder (
   Object folder); (Communicator 4.5 or later)
```

Parameters

The `SetPackageFolder` method has the following parameter:

`folder` An object representing a directory. You create this object by passing a string representing the directory to the `GetFolder` or `GetFolderComponent` method.

Returns

Nothing.

Description

The `SetPackageFolder` method to set the default package folder that is saved with the root node. When the package folder is set, it is used as the default for forms of `AddSubcomponent` and other methods that have an optional `localDirSpec` parameter.

You should only call this method once, and you should always call it immediately after you call `StartInstall`. If you call `SetPackageFolder` multiple times, the last folder set is the folder that is saved in the Client Version Registry and used as the default for other installations.

StartInstall

Initializes the installation of the specified software and version.

Method of

`SoftwareUpdate`

Syntax

```
int StartInstall (
   String package,
```

```
   VersionInfo version,
   int flags); (Communicator 4.0 or later)

int StartInstall (
   String package,
   String version,
   int flags); (Communicator 4.5 or later)

int StartInstall (
   String package,
   String version); (Communicator 4.5 or later)
```

Paramet rs

The `StartInstall` method has the following parameters:

`package` The Client Version Registry pathname for the software (for example: `Plugins/Adobe/Acrobat` or `/royalairways/RoyalPI/`). It is an error to supply a null or empty name.

The name can be absolute or relative. A relative pathname is relative to the Communicator namespace. A relative pathname must start with `plugins/`, to be relative to the plug-ins portion of that namespace or `java/download/`, to be relative to the Java portion. All other parts of the Communicator area of the registry are reserved for use by Netscape.

The Client Version Registry is a hierarchical description of the software registered for use with Communicator. The registry name provided here is not the location of the software on the machine, it is the location of information about the software inside the registry. This distinction is important when you add components with the `AddSubcomponent` method. For information on choosing an appropriate registry name, see "Positioning Software in the Client Version Registry" on page 33.

`version` A `VersionInfo` object (Communicator 4.0 or later) or a `String` (Communicator 4.5 or later). This parameter can be null, in which case, the software is installed without a version. If you do not supply a version, future visits to your web page may trigger unnecessary downloads, greatly annoying the user.

When `version` is a `VersionInfo` object, it represents the version of the package in the Client Version Registry.

When `version` is a `String`, it should be four integer values delimited by a period representing the version number, such as "4.2.1.1234".

`flags` An optional value that modifies the standard behavior of the user interface: `NO_STATUS_DLG` (to suppress the display of the dialog box that asks the user to confirm the installation) and `NO_FINALIZE_DLG` (to suppress the display of the progress bar during the `FinalizeInstall` method). The `NO_STATUS_DLG` flag is intended for use by script writers that provide their own user interface. Both values can be specifi d by performing a bitwise OR operation on them. If you do not specify the flags parameter, the security dialog box and the progress bar are displayed.

Returns

An integer error code. For a list of possible values, see "Return Codes" on page 102.

Description

The `StartInstall` method initializes the installation of the specified software. You must call this method immediately after the constructor. It is an error to call any other `SoftwareUpdate` methods before calling `StartInstall`.

After calling `StartInstall`, your script must call `AbortInstall` or `FinalizeInstall` before it finishes. If your script does not call `AbortInstall` or `FinalizeInstall`, Communicator will not be able to clean up prop rly after your script finishes.

Example

To start installation for th Royal Airways plug-in, you could use this code:

```
version = new netscape.softupdate.VersionInfo(3, 2, 1, 0);
su = new netscape.softupdate.SoftwareUpdate(this, "Royal Airways Plug-in");
err = su.StartInstall("/royalairways/RoyalPI/", version,
   netscape.softupdate.SoftwareUpdate.FULL_INSTALL);
```

Uninstall

Uninstalls a package that was previously installed by a SmartUpdate process.

Meth d of

`SoftwareUpdate`

Syntax

```
int Uninstall (
   String packageName);  (Communicator 4.5 or later)
```

Parameters

The `Uninstall` method has the following parameter:

`packageName` A string specifying the name of the package to be uninstalled.

Returns

An integer error code. For a list of possible values, see "Return Codes" on page 102.

Description

Th `Uninstall` method uninstalls a package that was previously installed by a SmartUpdate process. If a file is busy, Communicator waits until the file is no longer busy to remove it.

Trigger

A triggering script on a web page uses a `Trigger` object in downloading and installing software.

In Package

`netscape.softupdate`

Method Summary

`CompareVersion`	Compares the version of a file or package with the version of an existing file or package.
`ConditionalSoftwareUpdate`	Initiates the downloading and installation of the specified version of the specified software, if necessary.
`GetVersionInfo`	Returns a `VersionInfo` object representing the version number from the Cli nt Version Registry for the specified software or component.
`StartSoftwareUpdate`	Initiates the downloading and installation of the specified software.
`UpdateEnabled`	Indicates whether or not the SmartUpdate is enabled for this client machine.

CompareVersion

Initiates the downloading and installation of the specified version of software based on the results of a version comparison.

Method of

`Trigger`

Syntax

```
int CompareVersion (
   String registryName,
   VersionInfo version);  (Communicator 4.5 or later)

int CompareVersion (
```

```
   String registryName,
   String version); (Communicator 4.5 or later)

int CompareVersion (
   String registryName,
   int major,
   int minor,
   int release,
   int build); (Communicator 4.5 or later)
```

Parameters

The `CompareVersion` method has the following parameters:

`registryName` The pathname in the Client Version Registry for the component whose version is to be compared.

This parameter can be an absolute pathname, such as `/royalairways/RoyalSW` or a relative pathname, such as `plugsin/royalairway/RoyalSW`.

Note that the registry pathname is not the location of the software on the computer; it is the location of information about the software inside the Client Version Registry.

`version` A `versionInfo` object containing version information or a `String` in the format *major.minor.release.build*, where *major*, *minor*, *release*, and *build* are integer values representing version information.

Returns

If the versions are the same or if SmartUpdate is not enabled, this method returns 0. If the version of `registryName` is smaller (earlier) than `version`, this method returns a negative number. Otherwise, this method returns a positive number. In particular, this method returns one of the following values:

- -4: `registryName` has a smaller (earlier) major number than `version`.
- -3: `registryName` has a smaller (earlier) minor number than `version`.
- -2: `registryName` has a smaller (earlier) release number than `version`.
- -1: `registryName` has a smaller (earlier) build number than `version`.
- 0: The version numbers are the same; both objects represent the same version.
- 1: `registryName` has a larger (newer) build number than `version`.
- 2: `registryName` has a larger (newer) release number than `version`.
- 3: `registryName` has a larger (newer) minor number than `version`.
- 4: `registryName` has a larger (newer) major number than `version`.

The following constants can be used to check the value returned by `CompareVersion`:

```
int MAJOR_DIFF = 4;
int MINOR_DIFF = 3;
int REL_DIFF = 2;
int BLD_DIFF = 1;
int EQUAL = 0;
```

In Communicator 4.5, the following constants are defined and available for checking the value returned by `CompareVersion`:

```
Trigger.MAJOR_DIFF
Trigger.MINOR_DIFF
Trigger.REL_DIFF
Trigger.BLD_DIFF
Trigger.EQUAL
```

Description

The `CompareVersion` method compares the version of a installed file or package with a specified version.

If `registryName` is not found in the Client Version Registry or if `registryName` does not have version, `registryName` is assumed to have a version of 0.0.0.0.

If `registryName` represents a file, `CompareVersion` checks for the existence of the file. If the file has been deleted, its version is assumed to be 0.0.0.0.

ConditionalSoftwareUpdate

Initiates the downloading and installation of the specified version of the specified software.

Meth d f

Trigger

Syntax

```
Boolean ConditionalSoftwareUpdate (
   String url,
   String registryName,
   VersionInfo version,
   int mode); (Communicator 4.0 or later)

Boolean ConditionalSoftwareUpdate (
   String url,
   String registryName,
   String version,
   int mode); (Communicator 4.5 or later)

Boolean ConditionalSoftwareUpdate (
   String url,
   String registryName,
   String version); (Communicator 4.5 or later)

Boolean ConditionalSoftwareUpdate (
   String url,
   String registryName,
   int diffLevel,
   String version,
   int mode); (Communicator 4.5 or later)
```

Parameters

The `ConditionalSoftwareUpdate` **method has the following parameters:**

`url` A uniform resource locator specifying the location of the JAR file containing th software update.

`registryName` The pathname in the Client Version Registry for the software that may be updat d. The value of `registryName` should match the name passed to the `GetVersionInfo` method of the `Trigger` object and to the `StartInstall` method of th `SoftwareUpdate` object.

`version` A `versionInfo` object containing version information or a `String` in the format *major.minor.release.build*, where *major*, *minor*, *release*, and *build* are integer valu s representing version information.

`flag` One of:

`Trigger.DEFAULT_MODE`

`Trigger.FORCE_MODE`

`Trigger.SILENT_MODE`

`Trigger.FORCE_MODE | Trigger.SILENT_MODE`

If `flag` is missing, `Trigger.DEFAULT_MODE` is assumed.

For a description of the effect of the `flag` parameter, see the `StartSoftwareUpdate` method.

`diffLevel` A constant that indicates sensitivity to differences in versions. A combination of:

`Trigger.MAJOR_DIFF`

`Trigger.MINOR_DIFF`

`Trigger.REL_DIFF`

`Trigger.BLD_DIFF`

For example, if you specify `MAJOR_DIFF`, the update is triggered only if `version` is newer than the item in the Client Version registry in the first digit.

If you specify a negative value, the install is triggered if `version` is older than the item in the Client Version Registry in the specified digit. This is useful when you need to downgrade a component. When you specify a negative `diffLevel`, `flag` must usually be `Trigger.FORCE_MODE`.

If you do not specify a value for the `diffLevel` parameter, the most sensitive difference level is assumed (`Trigger.BLD_DIFF`).

Returns

False if the update is not necessary; otherwise, true. A return value of true does not indicate a successful update was finished, simply that it started.

Description

The `ConditionalSoftwareUpdate` method triggers the start of the software update for a particular version of a component. The software update starts only if there is not a later version of the software installed in the Cli nt Version Registry. Contrast this with the `StartSoftwareUpdate` method and the `ConditionalSoftwareUpdate` method.

If `componentName` represents a file, the disk is checked for the existence of `componentName`. If the file does not exist, the installation is triggered.

Example

This example triggers an update only if the software isn't already on the machine or if an earlier version is there:

```
version = new netscape.softupdate.VersionInfo(2,0,1,0);
trigger = netscape.softupdate.Trigger;

if ( trigger.UpdatedEnabled() )
   trigger.ConditionalSoftwareUpdate (
```

```
"http://royalairways/royalpkg.jar", "/royalairways/RoyalSW", version,
trigger.DEFAULT_MODE);
```

GetVersionInfo

Returns an object representing the version number from the Client Version Registry for the specified component. It is used in both trigger scripts and installation scripts.

M thod of

`Trigger`

Syntax

```
VersionInfo GetVersionInfo (
   String component);
```

Parameters

The `GetVersionInfo` method has one parameter:

`component` The name of a component in the Client Version Registry.

Returns

If SmartUpdate is disabled, this method returns `NULL`.

Otherwise, it returns a `VersionInfo` object representing the version of the component. If the component has not been registered in the Client Version Registry or if the specified component was installed with a null version, this method returns null.

Installing a component with a null version indicates that the component should always be updated when the opportunity arises.

Example

This code uses the `GetVersionInfo` method to determine which JAR file to download:

```
// Get the version number for the currently installed software
installed_version = netscape.softupdate.Trigger.GetVersionInfo(
   "plugins/royalairways/RoyalSW");

// If the installed version is null (that is, not installed)
// or less than version 3.0, do a complete install.
if (installed_version == null ||
    installed_version.compareTo(v3_0) < 0 )
      StartSoftwareUpdate("http://www.royalairways.com/v3_2.jar",
         trigger.DEFAULT_MODE);

// If the installed version is 3.0, do this update
else if (installed_version.compareTo(v3_1) < 0)
   netscape.softupdate.Trigger.ConditionalSoftwareUpdate (
      "http://www.royalairways.com/incremental_v0_to_v2.jar",
      "plugins/royalairways/RoyalSW", v3_2, trigger.DEFAULT_MODE);
```

StartSoftwareUpdate

Triggers the downloading and installation of the software at the specified URL.

M thod of

`Trigger`

Syntax

```
Boolean StartSoftwareUpdate (
```

SP

```
   String url,
   int flag); (Communicator 4.0 or later)

Boolean StartSoftwareUpdate (
   String url); (Communicator 4.5 or later)
```

Parameters

The `StartSoftwareUpdate` method has the following parameters:

`url` A uniform resource locator specifying the location of the JAR file containing the software.

`flag` One of:

`Trigger.DEFAULT_MODE`

`Trigger.FORCE_MODE`

`Trigger.SILENT_MODE`

`Trigger.FORCE_MODE | Trigger.SILENT_MODE`

`Trigger.FORCE_MODE | Trigger.SILENT_MODE`

If `flag` is missing, `Trigger.DEFAULT_MODE` is assumed.

The `flag` parameter is described in detail below.

Returns

True.

Description

The `StartSoftwareUpdate` method triggers a software update without first checking to see if a later version of the package exists. Contrast this with the `ConditionalSoftwareUpdate` method.

The `flag` parameter specifies information to pass to the installation script. There are two flags you can pass, `Trigger.FORCE_MODE` and `Trigger.SILENT_MODE`.

Passing the `Trigger.FORCE_MODE` flag requests that an installation be allowed to override a more recent version of a component. A trigger script can use this flag to request a component be downgraded. If `Trigger.FORCE_MODE` is included, a more recent version of a component can be overridden. It it is not included, a component is installed if there is no previously installed version or if the installed version number is null or smaller than the specified version.

For the request to be complied with, in the corresponding installation script, all calls to the `SoftwareUpdate` obj ct's `AddSubcomponent` method must pass `this.force` as their last parameter. `this.force` reflects the value of th `Trigger.FORCE_MODE` flag.

The `Trigger.SILENT_MODE` flag requests a silent installation. If this flag is included, the progress and permission dialog box may not appear while the software is being downloaded and installed. Once again, for the request to be complied with, the corresponding installation script must obey the request. The `Trigger.SILENT_MODE` flag is indicated in the installation script by the value of `this.silent`. If `this.silent` is true, the installation script should suppress the display of dialog boxes.

In addition, a silent installation can occur only if the signer of the JAR file has the `SilentInstall` privilege. This privilege can be set only using Netscape Mission Control.

The `SILENT_MODE` flag never suppresses the appearance of security dialog boxes.

Example

The following cod uses the `StartSoftwareUpdate` method to unconditionally trigger a download from `http://royalairways/royalpkg.jar` as long as SmartUpdate is enabled on the browser:

```
trigger = netscape.softupdate.Trigger;
if ( trigger.UpdatedEnabled() )
```

```
trigger.StartSoftwareUpdate (
   "http://royalairways/royalpkg.jar", trigger.DEFAULT_MODE);
```

UpdateEnabled

Indicates whether or not the JAR Installation Manager is enabled for this client machine.

Method f

`Trigger`

Syntax

```
Boolean UpdateEnabled (); (Communicator 4.0 or later)
```

Parameters

None

Returns

True if SmartUpdate is enabled for this client machine; otherwise, false. The method reflects the value of th `autoupdate.enabled` preference.

Example

The following code uses the `StartSoftwareUpdate` method to unconditionally trigger a download from `http://royalairways/royalpkg.jar` as long as SmartUpdate is enabled on the browser:

```
trigger = netscape.softupdate.Trigger;
if ( trigger.UpdatedEnabled() )
   trigger.StartSoftwareUpdate (
      "http://royalairways/royalpkg.jar", trigger.DEFAULT_MODE);
```

VersionInfo

You use `VersionInfo` objects to contain version information for software. This object and its methods ar used both when triggering a download, to see whether a particular version needs to be installed, and when installing the software.

In Package

`netscape.softupdate`

Method Summary

`VersionInfo`	Creates a `VersionInfo` object.
`compareTo`	Compares the version information specified in this object to the version information specified in the `version` parameter.

VersionInfo

Creates a `VersionInfo` object.

Method of

`VersionInfo`

Syntax

```
VersionInfo (
   int maj,
   int min,
   int rev,
   int bld); (Communicator 4.0 or later)
```

```
VersionInfo (
   String version); (Communicator 4.5 or later)
```

Parameters

The `VersionInfo` constructor has the following parameters:

`maj` The major version number.

`min` Minor version number.

`rev` Revision number.

`bld` Build number.

`version` A string representing version information in the format "4.1.2.1234". This parameter is available in Communicator 4.5 or later.

When maj, min, rev, and bld are provided as parameters, all four parameters are required, but all of them can be zero.

Returns

A new `VersionInfo` object.

Example

This code constructs a `VersionInfo` object for the 3.2.1 version of the Royal Airways plug-in using the integer form:

```
version = new netscape.softupdate.VersionInfo(3, 2, 1, 0);
```

This code constructs a `VersionInfo` object for the 3.2.1 version of the Royal Airways plug-in using the string form:

```
version = new netscape.softupdate.VersionInfo("3.2.1");
```

compareTo

Compares the version information specified in this object to the version information specified in the `version` parameter.

Method of

`VersionInfo`

Syntax

```
compareTo (
   VersionInfo version); (Communicator 4.0 or later)

compareTo (
   String version); (Communicator 4.5 or later)

compareTo (
   int major,
   int minor,
   int release,
   int build); (Communicator 4.5 or later)
```

Parameters

The `compareTo` method has the following parameters:

`maj` The major version number.

`min` Minor version number.

`rev` Revision number.

`bld` Build number.

`version` A `VersionInfo` object (Communicator 4.0 or later) or a `String` representing version information in the format "4.1.2.1234" (Communicator 4.5 or later).

Returns

If the versions are the same, this method returns 0. If this version object represents a smaller (earlier) version than that represented by the `version` parameter, this method returns a negative number. Otherwise, it returns a positive number. In particular, this method returns one of the following values:

- -4: This version object has a smaller (earlier) major number than `version`.
- -3: This version object has a smaller (earlier) minor number than `version`.
- -2: This version object has a smaller (earlier) release number than `version`.
- -1: This version object has a smaller (earlier) build number than `version`.
- 0: The version numbers are the same; both objects represent the same version.
- 1: This version object has a larger (newer) build number than `version`.
- 2: This version object has a larger (newer) release number than `version`.
- 3: This version object has a larger (newer) minor number than `version`.
- 4: This version object has a larger (newer) major number than `version`.

The following constants can be used to check the value returned by `CompareVersion`:

```
int MAJOR_DIFF = 4;
int MINOR_DIFF = 3;
int REL_DIFF = 2;
int BLD_DIFF = 1;
int EQUAL = 0;
```

In Communicator 4.5, the following constants are defined and available for checking the value returned by `compareTo`:

```
VersionInfo.MAJOR_DIFF
VersionInfo.MINOR_DIFF
VersionInfo.REL_DIFF
VersionInfo.BLD_DIFF
VersionInfo.EQUAL
```

Example

This code uses the `compareTo` method to determine whether or not version 3.2.1 of the Royal Airways software has been previously installed:

```
newVI = new netscape.softupdate.VersionInfo(3, 2, 1, 0);
existingVI = netscape.softupdate.Trigger.GetVersionInfo("/royalairways/royalsw");

if ( existingVI.compareTo(newVI) <= 0 ) {
   // ... proceed to update ...
}
```

WinProfile

(Windows only)

Windows developers use this object to manipulate the content of a Windows `.ini` file. This object does not have a public constructor. Instead, you construct an instance of this object by calling the `GetWinProfile` method of the `SoftwareUpdate` object.

In Package

`netscape.softupdate`

Method Summary

`getString` Retrieves a value from a `.ini` file.
`writeString` Changes a value in a `.ini` file.

getString

Retrieves a value from a `.ini` file.

Method of

`WinProfile`

Syntax

```
String getString (
   String section,
   String key);
```

Parameters

The method has the following parameters:
`section` Section in the file, such as `"boot"` or `"drivers"`.

`key` The key in that section whose value to return.

Returns

The value of the key or an empty string if none was found.

Description

The `getString` method is similar to the Windows API function `GetPrivateProfileString`. Unlike that function, this method does not support using a null key to return a list of keys in a section.

Example

To get the name of the wallpaper file from the desktop section of the `win.ini` file, use this call:

```
ini = su.GetWinProfile (su.GetFolder("Windows"), "win.ini");
ini.getString ("Desktop", "Wallpaper");
```

writeString

Changes a value in a `.ini` file.

Method of

`WinProfile`

Syntax

```
Boolean writeString (
   String section,
   String key,
   String value);
```

Parameters

The method has the following parameters:

`section` Section in the file, such as "`boot`" or "`drivers`".

`key` The key in that section whose value to change.

`value` The new value.

Returns

True if successfully scheduled, otherwise, false.

Description

The `writeString` method is similar to the Windows API function `WritePrivateProfileString`. To delete an existing value, supply null as the `value` parameter. Unlike the `WritePrivateProfileString` function, this method does not support using a null key to delete an entire section.

Valu s are not actually written until `FinalizeInstall` is called.

Example

To set the name of the wallpaper file from the desktop section of the `win.ini` file, use this call:

```
ini = su.GetWinProfile (su.GetFolder("Windows"), "win.ini");
ini.writeString ("Desktop", "Wallpaper", "newpathname");
```

WinReg

(Windows only)

Windows developers use this object to manipulate the content of the Windows registry. This object does not have a public constructor. Instead, you construct an instance of this object by calling the `GetWinRegistry` method of the `SoftwareUpdate` object.

This discussion assumes you are already familiar with the Windows Registry. For information on it, see API documentation for Windows NT or Windows 95.

When you construct a `WinReg` object, it is set to operate with `HKEY_CLASSES_ROOT` as its root key. To use a different root key, use the `setRootKey` method. Typically values in the Windows Registry are strings. To manipulate such values, use the `getValueString` and `setValueString` methods. To manipulate other values, use the `getValue` and `setValue` methods.

Reading registry values is immediate. However, writing to the registry is delayed until `FinalizeInstall` is called.

In Package

`netscape.softupdate`

Method Summary

`createKey`	Adds a key.
`deleteKey`	Removes a key.
`deleteValue`	Removes the value of an arbitrary key.
`getValue`	Retrieves the value of an arbitrary key.
`getValueString`	Retrieves the value of a key, when that value is a string.
`setRootKey`	Changes the root key being accessed.
`setValue`	Sets the value of an arbitrary key.
`setValueString`	Sets the value of a key, when that valu is a string

creat Key

Adds a key to the registry.

Meth d of

`WinReg`

Syntax

```
int createKey (
   String subkey,
   String classname);
```

Parameters

The method has the following parameters:

`subkey` The key path to the appropriate location in the key hierarchy, such as "`Software\\Netscape\\Navigator\\Mail`".

`classname` Usually an empty string. For information on this parameter, see the description of `RegCreateKeyEx` in your Windows API documentation.

R turns

0 if it succeeded; a nonzero number if it failed to schedule the creation. For a list of possible values, see "Return Codes" on page 102.

Description

The `createKey` method adds a key to the registry. You must add a key to the registry before you can add a valu for that key.

deleteKey

Removes a key from the registry.

Method of

`WinReg`

Syntax

```
int deleteKey (
   String subkey);
```

Parameters

Th method has the following parameters:

`subkey` The key path to the appropriate location in the key hierarchy, such as "`Software\\Netscape\\Navigator\\Mail`".

Returns

0 if it succeeded; a nonzero number if it failed to schedule the deletion. For a list of possible values, see "Return Codes" on page 102.

del teValue

Removes the value of an arbitrary key.

Method of

`WinReg`

Syntax

```
int deleteValue (
   String subkey,
   String valname);
```

Parameters

The `deleteValue` method has the following parameters:

`subkey` The key path to the appropriate location in the key hierarchy, such as "`Software\\Netscape\\Navigator\\Mail`".

`valname` The name of the value-name/value pair you want to remove.

Returns

0 if it succeeded; a nonzero number if it failed to schedule the deletion.

getValue

Retrieves the value of an arbitrary key.

Method of

`WinReg`

Syntax

```
WinRegValue getValue (
   String subkey,
   String valname);
```

Parameters

The `getValue` method has the following parameters:

`subkey` The key path to the appropriate location in the key hierarchy, such as "`Software\\Netscape\\Navigator\\Mail`".

`valname` The name of the value-name/value pair whose value you want.

Returns

A `WinRegValue` object representing the value of the named value-name/value pair or null if there is no value or if there is an error. See `WinRegValue` for information about these values.

Description

The `getValue` method retrieves the value of an arbitraty key. Use this method if the value you want is not a string. If the value is a string, the `getValueString` method is more convenient.

getValueString

Retrieves the value of a key, when that value is a string.

Method of

`WinReg`

Syntax

```
String getValueString (
   String subkey,
   String valname);
```

Parameters

The `getValueString` method has the following parameters:

`subkey` The key path to the appropriat location in the key hierarchy, such as "`Software\\Netscape\\Navigator\\Mail`".

`valname` The name of the value-name/value pair whose value you want.

Returns

A string representing the value of the named value-name/value pair or null if there's an error, the value is not found, or the value is not a string.

Description

The `getValueString` method gets the value of a string. If the value is not a string, use the `getValue` method instead.

setRootKey

Changes the root key being accessed.

Method of

`WinReg`

Syntax

```
void setRootKey (
   int key);
```

Parameters

The method has the following parameter:
`key` An integer representing a root key in the registry.

Returns

Nothing.

Description

The `setRootKey` changes the root key. When you create a `WinReg` object, it is set to access keys under the `HKEY_CLASSES_ROOT` portion of the registry. If you want to access keys in another portion, you must use this method to change the root key.

On 16-bit Windows platforms, `HKEY_CLASSES_ROOT` is the only valid value and this method does nothing.

These root keys are represented as fields of the `WinReg` object. The values you can use are:

- `HKEY_CLASSES_ROOT`
- `HKEY_CURRENT_USER`
- `HKEY_LOCAL_MACHINE`
- `HKEY_USERS`

Example

To use the `HKEY_USERS` section, use these statements:

```
su = new netscape.softupdate.SoftwareUpdate(this, "Royal Software");
winreg = su.GetWinRegistry();
winreg.SetRootKey(winreg.HKEY_USERS);
```

s tValue

Sets the value of an arbitrary key.

Method of

`WinReg`

Syntax

```
String setValue (
   String subkey,
   String valname,
   WinRegValue value);
```

Parameters

Th `setValue` method has the following parameters:

`subkey` The key path to the appropriate location in the key hierarchy, such as "`Software\\Netscape\\Navigator\\Mail`".

`valname` The name of the value-name/value pair whose value you want to change.

`value` A `WinRegValue` object representing the new non-string value. See `WinRegValue` for information about these values.

Returns

0 if it succeeded; a nonzero number if it failed to schedule the action. For a list of possible values, see "Return Codes" on page 102.

Description

The `setValue` method sets the value of an arbitrary key. Use this method if the value you want to set is not a string. If the value is a string, the `setValueString` method is more convenient.

setValueString

Sets the value of a key, when that value is a string.

Method of

`WinReg`

Syntax

```
int setValueString (
   String subkey,
   String valname,
   String value);
```

Parameters

The method has the following parameters:

`subkey` The key path to the appropriate location in the key hierarchy, such as "`Software\\Netscape\\Navigator\\Mail`".

`valname` The name of the value-name/value pair whose value you want to change.

`value` The new string value.

Returns

0 if it succeeded; a nonzero number if it failed to schedule the action. For a list of possible values, see "Return Codes" on page 102.

D scription

The `setValueString` method sets the value of a key when that value is a string. Use this method if the value you want to set is a string. If the value is not a string, us the `setValue` method instead.

WinR gValue

(Windows only)

Advanced Windows developers can use this object to manipulate non-string values for the Windows Registry. An object of this type has two fields: the type of the data and the value. For information on the possible data types for a registry value, see your Windows API documentation. You supply the value for these fields to the constructor for this class.

In Package

`netscape.softupdate`

WinRegValue

Creates a `WinRegValue` object.

Syntax

```
WinRegValue (
   int datatype,
   byte[] regdata);
```

Parameters

The `WinRegValue` constructor takes the following parameter:

`datatype` An integer indicating the type of the data encapsulated by this object. The possible values are:

- `WinRegValue.REG_SZ` = 1
- `WinRegValue.REG_EXPAND_SZ` = 2
- `WinRegValue.REG_BINARY` = 3
- `WinRegValue.REG_DWORD` = 4
- `WinRegValue.REG_DWORD_LITTLE_ENDIAN` = 4
- `WinRegValue.REG_DWORD_BIG_ENDIAN` = 5
- `WinRegValue.REG_LINK` = 6
- `WinRegValue.REG_MULTI_SZ` = 7
- `WinRegValue.REG_RESOURCE_LIST` = 8
- `WinRegValue.REG_FULL_RESOURCE_DESCRIPTOR` = 9
- `WinRegValue.REG_RESOURCE_REQUIREMENTS_LIST` = 10

`regdata` A Java byte array containing the data.

Returns

A new `WinRegValue` object, with the data members `type` and `data` set to the values passed to this constructor.

Return Codes

The methods described in this chapter can return any of the following return codes. In Communicator 4.5, thes constants are defined as part of the SoftwareUpdate object.

Name	Code	Explanation
`SUCCESS`	0	Success.
`REBOOT_NEEDED`	999	The files were installed, but one or more components were in us . Restart the computer and Communicator to complete the installation process. On Windows NT, you may only need to restart Communicator as long as you did not replace operating system files.
`BAD_PACKAGE_NAME`	-200	A probl m occurred with the package name supplied to `StartInstall`
`UNEXPECTED_ERROR`	-201	An unr cognized rror occurred.

ACCESS_DENIED	-202	The user did not grant the required security privilege.
TOO_MANY_CERTIFICATES	-203	Installation script was signed by more than one certificate
NO_INSTALLER_CERTIFICATE	-204	Installation script was not signed
NO_CERTIFICATE	-205	Extracted file is not signed or the file (and, therefor , its c rtificate) could not be found.
NO_MATCHING_CERTIFICATE	-206	Extracted file was not signed by the certificate used to sign the installation script
UNKNOWN_JAR_FILE	-207	JAR file has not been opened
INVALID_ARGUMENTS	-208	Bad parameters to a function
ILLEGAL_RELATIVE_PATH	-209	Illegal relative path
USER_CANCELLED	-210	User clicked Cancel on Install dialog
INSTALL_NOT_STARTED	-211	A problem occurred with the parameters to `StartInstall`, or `StartInstall` was not called first
SILENT_MODE_DENIED	-212	The silent installation privilege has not been granted.
NO_SUCH_COMPONENT	-213	The specified component is not present in the Client Version Registry.
FILE_DOES_NOT_EXIST	-214	The specified file cannot be deleted because it does not exist.
FILE_READ_ONLY	-215	The specified file cannot be deleted because its permissions are s t to read only.
FILE_IS_DIRECTORY	-216	The specified file cannot be deleted because it is a directory.
NETWORK_FILE_IS_IN_USE	-217	The specified file cannot be deleted because it is in use.
APPLE_SINGLE_ERR	-218	An error occurred when unpacking a file in AppleSingle format.
INVALID_PATH_ERR	-219	The path provided to `GetFolder` was invalid.
PATCH_BAD_DIFF	-220	An error occurred in GDIFF.
PATCH_BAD_CHECKSSUM_TARGET	-221	The checksum generated for the source file does not match the checksum in the JAR file.
PATCH_BAD_CHECKSUM_RESULT	-222	The checksum of the patched file failed.
UNINSTALL_FAILED	-223	An error occurred while uninstalling a package.

Appendix A
Sample Installation Script

This appendix contains a sample template for a JavaScript-based installation using the SmartUpdate technology.

This is the installation script used for the Netcaster component of Communicator. It is a good template for you to start with for your own installation.

```
// First some functions to use in the installation.

// Used only for debugging when execution needs to be
// slowed down to follow the output to the Java Console.
function delay(amount) {
   if( (debugOutput) && (amount > 0) )
   {
      var time = new Date();
      var seconds = time.getSeconds();
      var startCount = 80;
      var newSeconds = 70;

      startCount = (seconds + amount ) % 60;
      time = new Date();
      newSeconds = time.getSeconds();
      while (newSeconds != startCount ) {
         time = new Date();
         newSeconds = time.getSeconds();
      }
   }
}

// Conditionally displays a message only when debugging.
function dbgIfMsg(condition, message) {
   if(debugOutput && condition)
      java.lang.System.out.println(message);
}

// Displays a message only when debugging.
function dbgMsg(message) {
   if(debugOutput)
      java.lang.System.out.println(message);
}

// Pops an alert window only when NOT installing in silent mode.
function cAlert(message) {
   if(!this.silent)
      alert(message);
}

// Checks OS and version information.
function checkSystemEnvironment() {
   var err = 0;

   if(debugOutput) {
      registry_vi = netscape.softupdate.Trigger.GetVersionInfo("Netcaster");
      dbgIfMsg( (registry_vi == null ), "Warning: No registry info for Netcaster node");
      if (registry_vi != null) {
         dbgIfMsg( (vi.compareTo(registry_vi) <= 0),
                   "Warning: Version is no newer than previously installed version.");
      }
   }
```

```
    dbgMsg("The value of navigator.platform is: " + navigator.platform);
    if (navigator.platform != "Win32") {
        err = -20; // wrong OS
        dbgMsg("Error: Wrong operating system!");
        cAlert("Error: Wrong operating system: " + navigator.platform);
        return err;
    }
    return err;
}

// A wrapper for calling the AddSubcomponent() function.
function newSub(fName, jarFilePath, tgtVI, tgtFolder, tgtFilePath) {
    dbgMsg("Filename in newSub is: " + jarFilePath);
    var err = su.AddSubcomponent(fName,tgtVI,jarFilePath,tgtFolder,tgtFilePath,this.for
    dbgIfMsg( (err != 0), "Error Adding SubComponent " + fName + "error value: " + err);
    if(err != 0) {
        newSubFailure = true;
    }
    delay(FileDelayTime);
}

// This prepares the specific files to be installed.
function setupFiles(su) {
    var err = 0;

    if (su == null) {
        dbgMsg("Error passing su to setupFiles: " + err);
        return -1;
    }

    err = su.StartInstall("Netcaster",          // Package name
                          vi,
                          netscape.softupdate.SoftwareUpdate.FULL_INSTALL);

    if (err !=0) {
        dbgMsg("Error at startInstall: " + err);
        return err;
    }

    // Get folders.
    CommFolder = su.GetFolder("Communicator");
    if (CommFolder == null) {
        dbgMsg("ERROR GETTING FOLDER: Communicator");
        return -1;
    }
    JClassFolder = su.GetFolder("Netscape Java Classes");
    if (JClassFolder == null) {
        dbgMsg("ERROR GETTING FOLDER: Netscape Java Classes");
        return -1;
    }
    HelpFolder = su.GetFolder("Program");
    if (HelpFolder == null) {
        dbgMsg("ERROR GETTING FOLDER: Program");
        return -1;
    }

    // Add subcomponents.
    //   Netcaster files
    newSub("", "admin.jar",     vi, CommFolder, "Netcast/admin.jar");
    newSub("", "ncjava10.jar", vi, CommFolder, "Netcast/ncjava10.jar");
    newSub("", "ncjs10.jar",    vi, CommFolder, "Netcast/ncjs10.jar");
    newSub("", "addc.htm",      vi, CommFolder, "Netcast/addc.htm");
    newSub("", "images/blank.gif",    vi, CommFolder, "Netcast/images/blank.gif");
    // ... more files ...
    );

    //   Java Classes
    newSub("marimb10.jar", "JavaClasses/marimb10.jar", vi, JClassFolder, "marimb10.jar");
```

```
    //   Netcaster Help files
    newSub("=COMM=/NetHelp/Netscape/netcastr/help.hpf",     "Help/help.hpf",     vi, He
    newSub("=COMM=/NetHelp/Netscape/netcastr/net.htm",      "Help/net.htm",      vi, He
    newSub("=COMM=/NetHelp/Netscape/netcastr/netHdr.htm",   "Help/netHdr.htm",   vi, He
    newSub("=COMM=/NetHelp/Netscape/netcastr/netcastr.gif", "Help/netcastr.gif", vi, He
    newSub("", "nc_startup.html", vi, HelpFolder, "Netcast/nc_startup.html");

    // If any subcomponent failed, the installation aborts here.
    if (newSubFailure) {
       dbgMsg("Error adding at least one subcomponent.");
       abortMe();
     }

    return 0;
}

// Handles catastrophic errors.
function abortMe(err) {
    if(!abortCalled) {
       dbgMsg("Install Aborted." + err);
       cAlert("Install Aborted." + err);
       su.AbortInstall();
       abortCalled = true;
    }
}
// End of functions.

// ==================== START ACTUAL INSTALLATION ========================
//
// Global variable declarations
//
var updateObjectName = "Netscape Netcaster v1.0 Install";
var versionMaj = 4;
var versionMin = 0;
var versionRel = 4;
var versionBld = 97139;
var FileDelayTime = 0; // Number of seconds delay between subcomponents when debugging

var abortCalled   = false;
var newSubFailure = false;
var debugOutput   = true;  // Turns debugging output on/off

var vi = new netscape.softupdate.VersionInfo(versionMaj, versionMin, versionRel, versi
    dbgIfMsg( (vi == null), "Warning: Unable to create the VersionInfo object.");
var su = new netscape.softupdate.SoftwareUpdate( this, updateObjectName );
    dbgIfMsg( (su == null), "Warning: Unable to create the SoftwareUpdate object.");

//--------------------------------------------------
// Here starts the main body of execution.
//--------------------------------------------------
java.lang.System.out.println("Starting script...");

var err = 0;
if ( (su != null) ) {
   dbgMsg("new SoftwareUpdate succeeded.");
   err = checkSystemEnvironment();
   if ( err == 0 ) {
      dbgMsg("checkSystemEnvirnoment succeeded.");
      err = setupFiles(su);
      if (err == 0) {
         dbgMsg("setupFiles succeeded.");
         err = su.FinalizeInstall();              // This actually copies all the files.
         if (err == 0) {
            cAlert("Install Complete: Relaunch the Navigator to enable
                    Netcaster menu selection.");
            cAlert("Install Successfully Completed");
```

```
            }
            else {
                dbgMsg("Error at FinalizeInstall: " + err);
                abortMe(err);
            } //FinalizeInstall
        }
        else {
            dbgMsg("Error at setupFiles: " + err);
            abortMe(err);
        } //setupFiles
    }
    else {
        dbgMsg("Error at checkSystemEnvironment: " + err);
        abortMe(err);
    } //checkSystemEnvironment
}
```

Appendix B
End Us r Problems

This appendix contains information on problems an end user might encounter when trying to install software using SmartUpdate. These errors all occur after Communicator has downloaded the JAR archive to the end user's machine. For additional information about resolving SmartUpdate end user problems, see http://help.netscape.com/kb/netcenter/971023-6.html.

Communicator Prompts to Save JAR as an .exe File

Caus : The user has attempted to directly download a JAR archive and is accessing the software through a proxy. That proxy has assigned an incorrect MIME type to a JAR archive.

Fix/w rkaround: The content provider should use a trigger script to start the download of the JAR archive, inst ad of having the user directly download the JAR archive. In a trigger script for Communicator 4.0.2 or later, Navigator treats any downloaded file (other than an HTML file) as a JAR archive.

To work around the problem, the user can save the file to the disk, give it a `.jar`extension, and open it locally in Communicator.

C mmunicator Displays the Message: Downloaded file is not a JAR archive

Cause There are several possible causes of this error:

- The file that SmartUpdate got from the server was not a JAR archive. The server might have returned an HTML error message (for example, if it is too busy, or the file was not available).
- The server is misconfigured, and is serving JAR archives with the wrong MIME type.
- The user is accessing the file through a proxy, and the proxy is misconfigured for the MIME type of JAR archives.

Fix/workaround: There are also several possible fixes and workarounds:

- Make sure that the file is available on the server.
- Make sure that the server is configured properly. (Files with the `.jar`extension should be served as `application/java-archive`.)
- The user can save the file to a local disk (with a suffix of `.jar`) and open it inside Communicator.
- The developer can make sure that the MIME type of the JAR archive is propagated correctly by serving files off an HTTP server rather than an FTP server.
- The content provider can use a trigger script to start the download instead of directly accessing the JAR archive. In a trigger script for Communicator 4.0.2 or later, Navigator treats any downloaded file (other than an HTML file) as a JAR archive.

C mmunicator Displays the Message: SmartUpdate failed: JAR archive failed a security check.

Cause: The signature inside the JAR archive did not pass the security check. The possible reasons for this are numerous. For more details refer to the information on security at Security Developer Central. Some possible problems include:

- The JAR archive was corrupted during download.
- The certificate authority the developer us d to sign the archive is not recognized by Communicator. For exampl , if the developer used a company's certificate server to get the signing certificates, outside users probably will not trust that Certificate Authority.
- Navigator's security database is corrupted.

Fix/w rkar und: You have several options to try to fix the problem:

- Try downloading the file again
- Before starting SmartUpdate, show the user how to obtain your certificate authority. Or sign your code with a certificate issued by one of the built-in certificate authorities, such as VeriSign.
- Reinitialize your security database. Reinitializing the security database is an extremely high-risk operation. Do not do it unless you're certain it's necessary.

Table of Contents | Previous | Next | Index

Last Updated: 03/11/99 11:37:22

Appendix C
R leas Notes

This appendix contains release notes for SmartUpdate technology in Communicator 4.5.

- New Methods

The following changes have been made to the SoftwareUpdate class:

- `AddDirectory` (new method) (available in Communicator 4.05 and later, but not in Communicator 4.0 through 4.04)
- `AddSubcomponent` (new forms to simplify script writing)
- `DeleteFile` (new method)
- `DeleteComponent` (new method)
- `Execute` (overloaded form that allows command-line arguments on Windows and Unix systems)
- `GetFolder` (new forms that allow the specification of subdirectories)
- `GetComponentFolder` (new forms that allow the specification of subdirectories)
- `GetFolder` (new targets)
- `DiskSpaceAvailable` (new method)
- `GetLastError` (new method)
- `Patch` (new method, supported by a new tool, nsdiff)
- `ResetError` (new method)
- `SetPackageFolder` (new method)
- `StartInstall` (security privilege changes and new modes)
- `Uninstall` (new method)

The following changes have been made to the Trigger class:

- `CompareVersion` (new method)
- Difference level constants added from the VersionInfo class
- `StartSoftwareUpdate` (mode parameter is now optional)
- `ConditionalSoftwareUpdate` (new forms to simplify checking for version differences)

The following changes have been made to the VersionInfo class:

- Definition for difference level constants
- `The VersionInfo` constructor now takes a single string of the form "4.0.2.1234" in addition to the form that requires four integers.
- `compareTo` now allows script writers to compare versions without having to create a VersionInfo object.
- Security Privileg Changes

Prior to Communicator 4.5, the `StartInstall` method could be called with a `permission` parameter that could be `FULL_INSTALL` or `LIMITED_INSTALL`. In addition, the `SoftwareInstall`, `SilentInstall`, and `Uninstall` security privileges were defined as part of the `SoftwareUpdate` class.

Communicator 4.5 provides a form of `StartInstall` that eliminates the `permission` parameter. When you use this form of `StartInstall`, it assumes `FULL_INSTALL`. For backward compatibility, `LIMITED_INSTALL` can still be used with the `StartInstall` method, but you are encouraged to use the new form of `StartInstall` that does not require a `permission` parameter.

In Communicator 4.5, `SoftwareInstall`, `SilentInstall`, and `Uninstall` are normal security privileges defined in the Security Manager. With this change, any Java or JavaScript can request these privileges and bring up the security dialog box. This may be useful for web pages to get the privilege approved before the download.

If a trigger script requests `SILENT_MODE` and Communicator is not configured to support `SILENT_MODE`, Communicator 4.5 displays the standard security dialog box instead of the "Silent Install" security dialog box. This behavior is in contrast to previous versions of Communicator that displayed the "SilentInstall" dialog box in this circumstance.

Obtaining `SoftwareInstall` privilege includes a grant of `StandardRegistryAccess` so that items can be placed in the Shared or Private areas of the Client Version Registry for use by the application. If items are place in the Private area of the registry, the install script must be signed by the same certificate as the class that will eventually read from the registry.

Obtaining `SoftwareInstall` privilege also grants the `UniversalPreferencesRead` privilege, but it does not grant the `UniversalPreferencesWrite` privilege. Script writers who need `UniversalPrefernces Write` privilege must ask for that privilege on their own.

Obtaining `SilentInstall` privilege automatically grants `SoftwareInstall` privilege and all of its sub-privileges.

- Signed JavaScript

  In Communicator 4.5, SmartUpdate installation scripts are signed JavaScript programs that can use features that are available to any signed JavaScript program.

- Multiple Downloads

  Communicator 4.5 queues triggered installations to prevent multiple downloads that could overwhelm the network and any particular computer. Multiple installatons are downloaded and performed in the order in which they were triggered.

- Security Policy

  The JavaScript `navigator` object has a new, read-only `securityPolicy` property that is a string describing the security policy of the running Communicator or Navigator executable.

- Registry Nodes

  All SoftwareUpdate methods that take a registry name as a parameter can now take registry names that begin with "`=USER=/`" to refer to items registered under the current user. Methods that take a registry name as a parameter do not prepend the package registry name.

Appendix D
The NSDiff Utility

This appendix describes the `NSDiff` utility, which you use to create a file containing the differences between an existing component and an update of that component. You use the differences file for the purposes of using the `Patch` method of the `SoftwareUpdate` object.

The `NSDiff` utility can be downloaded from http://devedge.netscape.com/software/tools/SmartUpdate.html. Ther are versions of the utility for Windows 95/98/NT, Mac OS, and the Unix operating system.

The syntax for the `NSDiff` utilility is as follows:

`NSDiff [-bx] [-cx] [-d] [-wb-] -o"filename" oldfile newfile`

The command line options for the `NSDiff` utility are listed below:

`-a`	Specifies that *`oldfile`* and *`newfile`* are in AppleSingle format. (Mac OS version only)
`-bx`	Specifies in bytes the block size to use for the comparison. The default block size is 64. The minimum block size is 9. A smaller block size may result in a smaller differences file that tak s less time to download but more time to apply. A larger block size may result in a larger differences file that takes more time to download but takes takes less time to apply.
`-cx`	Specifies the checksum type. The default checksum type is CRC-32, which is the only supported checksum type at this time.
`-d`	Causes `NSDiff` to display diagnostic information while it executes.
`-o"filename"`	Specifies the name of the differences file. The default is *`newfilename`*`.gdf`
`-wb-`	Improves processing time by turning off special handling for Windows executables that have been processed by `BindImage`. (Windows 95/98/NT version only.)

Index

A

C

D

E

F

G

I

J

N

P

R

S

T

U

V

W

Z